Aalto University
School of Science

Maria Phan

Web Application Programming Interface Design for a Customer Portal

Master's Thesis

Espoo, November 21, 2015

Supervisor: Professor Eljas Soisalon-Soininen

Advisor: M.Sc. (Tech) Antti Tuomi

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	Maria Phan		
Title:	Web Application Programming Interface Design for a Customer Portal		
Date:	November 21, 2015	**Pages:**	x + 76
Minor:	Software Technology	**Code:**	T-220
Supervisor:	Professor Eljas Soisalon-Soininen		
Advisor:	M.Sc. (Tech) Antti Tuomi		

Home builders in the home building industry would like to increase customer satisfaction, by providing a self-service customer portal. In a customer portal, all stakeholders such as home builders, homeowners and vendors can meet and communicate and share distributed data. Customer self-service would reduce the overhead of customer service. Direct communication through portals could also prevent costly mistakes before they happen. Web Application Programming Interface (Web API) could provide a framework for builders to brand their own customer portal website.

A Web API could provide an interface where both components server and client side could evolve independently. A good API makes it easier to develop software and provides a building component for the application.

The purpose of this thesis is to design Web APIs for a customer portal in the home building industry. This thesis presents a design of a Web API, which concentrates on the architectural design of a Web API, including versioning and security. This design provides resource APIs, with Uniform Resource Identifier (URI) versioning and token based SHA-256 message authentication. The implementation of Web APIs allows the home provider to customize their own customer portal according to their own brand.

Keywords:	Web API, REST, HTTP, URI, Web Architecture, CRUD, Versioning, HMAC, Customer Portal
Language:	English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Maria Phan		
Työn nimi:	Web -ohjelmointirajapinnan suunnittelu asiakasportaalille		
Päiväys:	21. marraskuuta 2015	**Sivumäärä:**	x + 76
Sivuaine:	Ohjelmistotekniikka	**Koodi:**	T-220
Valvoja:	Professori Eljas Soisalon-Soininen		
Ohjaaja:	Diplomi-insinööri Antti Tuomi		

Kodin rakennuttajat haluaisivat lisätä asiakastyytyväisyyttä tarjoamalla itsepalveluportaalia. Asiakasportaalissa kaikki sidosryhmät, kuten rakennuttajat, kodin omistajat ja myyjät, voivat kommunikoida suoraan keskenään ja jakaa informaatiota. Itsepalvelu vähentäisi taakkaa asiakaspalvelussa. Suora kommunikointi portaalin kautta ehkäisisi vakavat virheet etukäteen. Web-ohjelmointirajapinta (Web API) tarjoaisi rakennuttajille välineet muokata Web-portaaliaan yhtiönsä brändin mukaisesti.

Web API tarjoaisi rajapinnan, missä sekä asiakaspuoli että palvelinpuoli voivat kehittyä itsenäisesti. Hyvä API-design helpottaa ohjelmiston suunnittelemista ja tarjoaa rakennuselementin sovelluksen kehittämiselle.

Tämän diplomityön tarkoitus on suunnitella Web API -asiakasportaali kodin rakennuttajille. Tässä tutkielmassa esitellään Web API -arkkitehtuuriin design, joka sisältää versioinnin ja truvallisuuden suunnittelua. Tämä malli tarjoaa resurssikeskeisen Web API:n, joka käyttää URI-versiointia ja Secure Hash -algoritmiin perustuvaa viestin autentikaatiota.

Asiasanat:	Web API, REST, HTTP, URI, Web -arkkitehtuuri, CRUD, versiointi, HMAC, asiakasportaali
Kieli:	Englanti

Acknowledgements

I am very grateful to CEO of Kova Finland OY Jouko Väkiparta for providing this amazing possibility to write the thesis. I am deeply grateful to my supervisor, Professor Eljas Soisalon-Soininen for being my supervisor and for guidance and ideas in writing this thesis. I thank my instructor M.Sc. (Tech) Antti Tuomi for valuable feedbacks and insights and also for the patient, helpful, and concrete comments on my thesis draft.

I am grateful to my colleagues for making Kova Finland Oy a great place to work. It is full of laughter and an inspirational place to learn while working.

Thank you, Hanna Hirvonen, Paula Puupponen, and Hanna Kaskela, for supporting me on my thesis writing process. Thank you, Elizabeth Pekkarinen, for helping me with the English grammar. Thank you, Heidi Pentikäinen, for giving feedbacks for the Finnish abstract.

I would like to thank my parents Phan Si Trang and Nguyen Thi Oanh, for being enthusiastic and creative for all new things that are shaking the world. Thank you for being an example of never giving up or being afraid of failures.

I want to express my gratitude to four persons who has deeply influenced me in my life with their endless wisdom and kindness. My very dear friend who has recently passed away, Helena Viherhaka, my Vietnamese teacher, Nguyen Kim Quyen, my awesome viola teacher, Pekka Jylhä and last but not least, Heikki Pekkarinen, who is like a father to me.

I dedicate this work to my loving and supporting husband, Tuukka Eeronheimo, who has been encouraging me patiently all this time. I thank my son, Elia, for being my little firefighter.

Thank you, God, for giving me the peace and wisdom to write this thesis and to survive this hard work. Whatever I do I want to do all things for your glory. Yay yay yay \0/

Hämeenlinna, November 21, 2015

Maria Phan

Abbreviations and Acronyms

API	Application Programming Interface
CRUD	Create, Retrieve, Update and Delete
CST	Customer Service Team
FIPS	Federal Information Processing Standard
HMAC	Hash-based message authentication code
HATEOAS	Hypermedia as the Engine of Application State
HTTP	Hypertext Transfer Protocol
IoT	Internet of Things
JSON	JavaScript Object Notation
MAC	Message authentication code
MD5	Message Digest 5
NIST	National Institute of Standards and Technology
REST	Representational State Transfer
RMM	REST Maturity Model
RPC	Remote Procedure Calls
SHA	Secure Hash Algorithm
URI	Uniform Resource Identifiers
URL	Uniform Resource Locator
URN	Uniform Resource Name
XML	Extensible Markup Language
W3C	The World Wide Web Consortium
WWW	World Wide Web

Contents

List of Figures

Chapter 1

Introduction

The use of self-service technologies is increasing across industries and there has been a big shift in industries developing self-service web portals [64] [57]. Self-service technologies usage will continue to grow because it reduces labor costs and increases customer satisfaction [23].

In order to increase satisfaction with customer service, home builders in the home building industries want to provide a self-service customer portal. Via a portal, homeowners can be in contact with any concerns 24/7 from all around the world. Direct communication between customers and customer services through portals may prevent costly mistakes before they happen [11]. Instead of using a general customer portal, with Web-based application interfaces (Web APIs) each home builder can customize and brand their own portal.

A Web API [2] would provide an interface where both components server and client side could evolve independently. Web API is a software interface exposed on the Web using Hyper Text Transfer Protocol (HTTP) and is used for developing Web applications accessible via the website. Web APIs give freedom and flexibility in customizing a Web application.

1.1 Goals of the Thesis

The purpose of this thesis is to provide customer portal Web APIs for home builders, that allow the client side application to evolve independently from the server side. The characteristics of the design of a customer portal Web APIs are evaluated. The design of the Web APIs has three design goals.

First, in order to let both client and server side to evolve independently, the degree of dependency between both elements should be minimized. How to design a Web API that keeps a client and a server separate?

Second, like any software, in time Web APIs are going to evolve and change to meet the changing needs of customers. How to provide versioning in the evolving Web API?

And thirdly, security is an important part of a design. Web APIs provide resource access via the interface. How to secure the access and let only limited access to Web API resource?

1.2 Structure of the Thesis

The remaining chapters of this thesis are organized as follows:

Chapter 2 explains what Web Application Programming Interfaces (API) are and gives a description of Web APIs for customer portal.

Chapter 3 introduces the Representational State Transfer (REST) architecture. Web APIs that adhere to REST architectural style are called RESTful Web APIs.

Chapter 4 shows how the RESTful style is applied to Web APIs according to four steps of Richardsons REST Maturity Model [61].

Chapter 5 introduces two ways of versioning Web APIs: a Uniform Resource Locator (URL) way and a Hypermedia way.

Chapter 6 describes security and validation used in Web APIs. Hash-based

Message Authentication (HMAC) can be used in authenticating each Web API credentials.

Chapter 7 presents home building industry in the USA and introduces Sapphire Build and how it is related to Customer Portal. In addition, it presents requirements for different customer portal users: lead, prospect, customer under contract and homeowner.

Chapter 8 describes the implementation of the Web API for a customer portal.

Chapter 9 evaluates the design of the Web API for a customer portal. Challenges and future work are discussed.

Chapter 10 summarizes the design and result of this thesis.

Chapter 2

Web Application Programming Interface

First, this chapter introduces Application Programming Interfaces (API). Then a brief description of different Web API architectural styles is given. Finally, we discuss why from the two dominant architectural approaches the Representational State Transfer (REST) style is more popular than the SOAP standard.

API provides an interface for developers to build a software application for colleagues, partners, or third-party developers [42]. It allows developers to access data and services to build applications. Famous examples of applications are Facebook, Twitter, and Youtube [59]. APIs can be accessible to any developer, only visible for partners or used privately between teams. In this thesis, the APIs are only visible for partners.

2.1 Web API versus a Website

Web API is a software interface exposed to the Web over Hyper Text Transfer Protocol (HTTP) [2]. Web APIs are used for developing Web applications accessible via the website.

A website provides consumable information such as news and blogs. A website can be altered, but all the changes are visible to users. Instead, an API has a contract and its programs are built on top of that contract, i.e., a developer cannot alter anything in the contract of the API without violating the applications built on top of it. However, this does not hold for implementation. The interface remains consistent, whereas the implementation can be changed on a daily basis. Similar to websites APIs are expected to be available 24/7. An API should be treated like a software product. However, this does not mean that the API never changes. If a change in an API breaks an application built on top of that, it is called a breaking change. There should be maintainability such as versioning and backward compatibility for necessary breaking changes. In this way, the API can keep up with a growing business and meet the changing need of customers [42].

2.2 Web API Architecture

Let's consider a basic architecture without an API where each client application has its own business logic, which connects directly to a database. This is illustrated in Figure 2.1. Business logic layer carries out the operations between a server side and the user interface [4]. Each client's business logic needs to be maintained in synchronization with each other when providing new features. This makes maintainability and evolvability very expensive since each application needs to be updated accordingly [8].

Figure 2.2 depicts the same architecture with a central API which has the same business logic for all applications. Each client uses the same API to get, create, update and delete resources. The change in business logic is made only in one place [8].

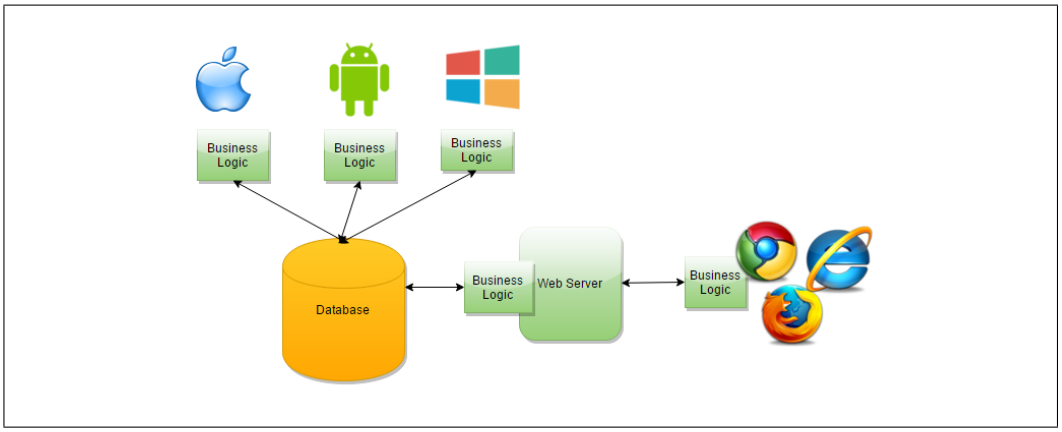

Figure 2.1: Web service without Web API, adapted from [8].

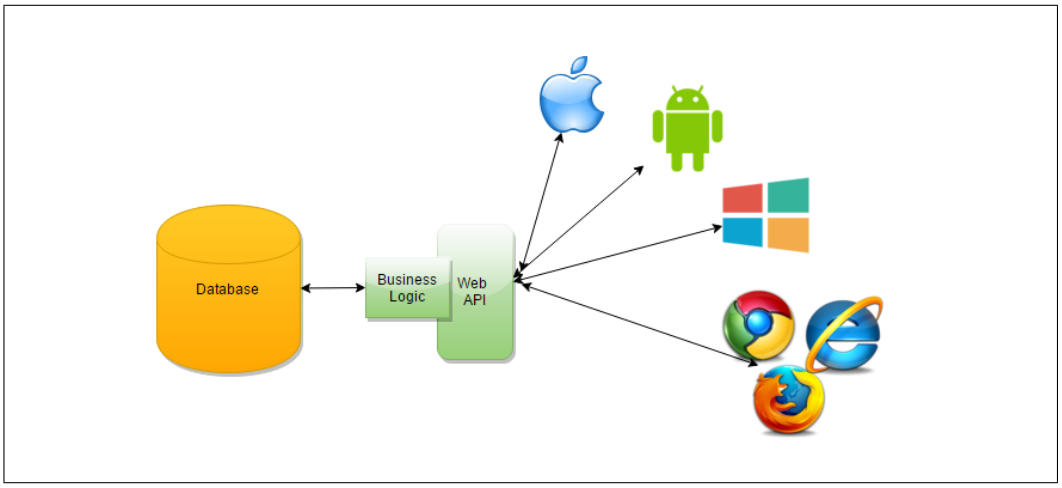

Figure 2.2: Web service with Web API, adapted from [8].

2.3 Web API Architectural Styles

An architectural style is not a standard. Moreover, it defines the set of style and characteristics. Take, for example, a Gothic cathedral, where a cathedral is built in Gothic style. By observing various buildings, one can determine whether or not the building has certain qualities of Gothic style such as pointed arches, rib vault, and flying buttresses [20]. In the same way, architectural styles for Web API have certain qualities and characteristics. The main Web API design styles are tunneling style, URI style, and REST architectural style [2].

2.3.1 Tunneling

A well-known implementation of tunneling style is SOAP protocol used in the remote procedure call (RPC) style [46]. It can use HTTP as a transport protocol. However, SOAP is a transport protocol agnostic and does not make use of HTTP verbs [46].

2.3.2 Uniform Resource Identifier

The Uniform Resource Identifier (URI) style is a resource-centric API, where each noun is considered as a resource. Resources are Created, Retrieved, Updated or Deleted (CRUD) [63]. CRUD operations are mapped with HTTP methods GET, POST, PUT, and DELETE respectively. Each resource is identified with unique URI. More details about CRUD and URIs are presented in section 4.1.3.

2.3.3 Representational State Transfer

The REST style is similar to the URI style. REST utilizes hypermedia to create interactions focused on tasks instead on resources [63]. More details about REST will be presented in Chapter 3.

Since the URI style can be considered as a substyle of REST, in the next section we can focus on differences between REST and SOAP.

2.4 Web API as Web Services: REST vs SOAP

According to the World Wide Web Consortium (W3C) [65], a web service is a software system designed to support communication between interoperable machines over a network. There are several approaches to web services. The two most dominant models for designing Web services are REST style web services and SOAP [56].

There has been a lot of research regarding the advantages and the disadvantages of REST and SOAP as Web services [43] [36] [49] [56] [55]. One of the

most common questions in debates is "Which is better, REST or SOAP?". However, they provide totally different approaches. REST is an architectural style [32] for building client-server applications. SOAP is a protocol specification for exchanging data between different elements [65].

Since REST is a Web architectural style and SOAP is a standard, it is more reasonable to compare REST with the remote procedure call (RPC) style of building client-server applications [43]. Though, SOAP does not require the RPC style.

As opposed to SOAP protocol, REST does not define a standard nor give definite guidelines to developers. It is more of a design approach for Web architecture. REST style is derived from several Network-based architectural style. The web service that adheres to the REST architectural style is called a RESTful web service. REST is used in accessing resources through a single consistent interface i.e. API [32].

In fact, one of the main characteristics and constraints of REST is a uniform interface constraint. That is, REST considers any kind of information as a resource, which can be identified with unique Uniform Resource Identifiers (URI). When REST is applied on HTTP[1], it uses HTTP methods GET, POST, PUT and DELETE for resource manipulation. More about the uniform interface constraint will be described in section 3.2.4.

SOAP builds several layers on the top of HTTP. Unlike REST, SOAP does not take advantage of HTTP possibilities and functionality. SOAP uses only the HTTP POST verb. Thus, SOAP responses cannot be cached with unsafe POST verb. Whereas with REST, the GET requests can be cached, which can be used to increase scalability [43].

REST uses many data formats such as JavaScript Object Notation (JSON) and eXtensible Markup Language (XML), whereas SOAP permits only XML [60] . JSON is generally used as data presentation [32]. In addition, JSON is also a subset of JavaScript, which is used to build web applications. More about JSON will be presented in section 3.2.7.2.

REST provides a lower barrier to implementing services and is more flexible to develop [56]. In web services coupling means an ability to modify the

[1]Note that, both REST and SOAP are protocol independent, since this thesis provides Web API design, it will focus on HTTP as a transfer protocol.

web service without affecting the clients. Under this definition, SOAP-based services are more tightly coupled compared to the REST. In loose coupling, the client depends on the API and the service contract. It does not require a change on the client side when there is a change on the service side [52].

SOAP implementations may have overhead on the network bandwidth consumption and execution time. Complexity makes it difficult for developers to change SOAP APIs and services without great code modifications [56]. REST is a simpler, more lightweight solution and, therefore, provides better performance [49]. Nevertheless, due to the simplicity of the REST approach, it might not be the best choice for operationally and functionally very complex systems [18]. Both REST and SOAP can be used to implement the same service, but, in general, SOAP should be used when a particular feature of SOAP is needed. However, the ease-of-use of REST makes it more desirable to adopt [43]. As we can see in figure 2.3 REST is more popular than SOAP.

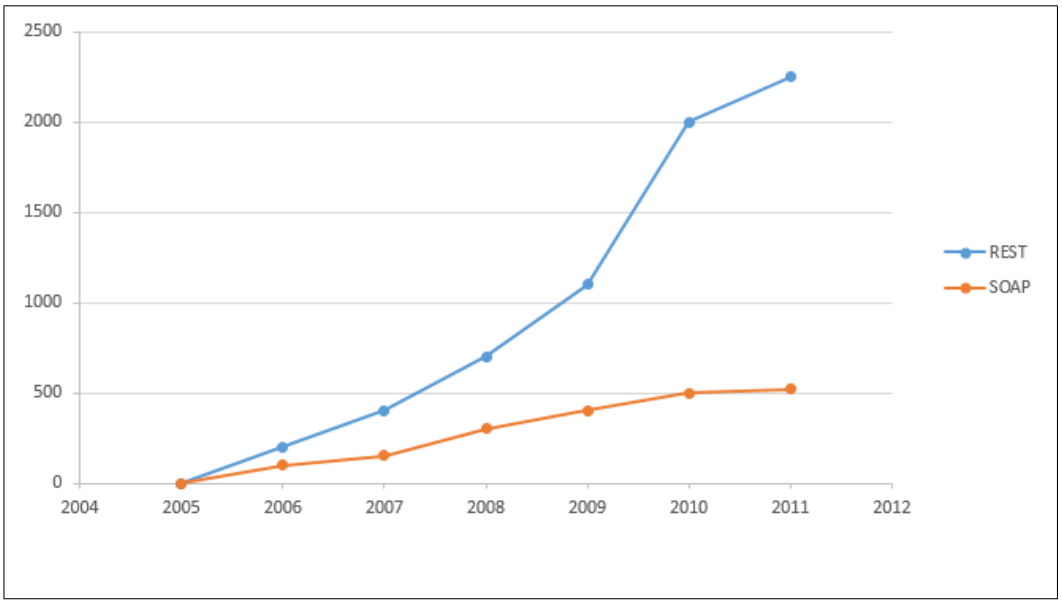

Figure 2.3: Distribution of APIs: REST versus SOAP, adapted from [59].

2.5 Summary

This chapter has provided the rationale for choosing a RESTful web API approach for a customer portal. The next chapter will go into details of REST answering these questions: what is REST and what are the characteristic of it?

Chapter 3

Representational State Transfer (REST)

This chapter introduces the Representational State Transfer (REST) style, it gives a brief background on REST and describes how REST architectural style works. Web APIs that adhere to REST style are called RESTful web APIs.

3.1 REST as Coined by Roy Fielding

Representational State Transfer (REST) is an architectural style for distributed hypermedia systems such as World Wide Web (WWW) [32]. REST was coined in 2000 by Roy Fielding in his Ph.D. dissertation "Architectural Styles and the Design of Network-based Software Architectures" at the university of California, Irvine. Fielding had a desire to understand and evaluate the architectural design of WWW [33]. REST architectural style was developed as an abstract model of the Web architecture to guide redesign and definition of the Hypertext Transfer Protocol (HTTP) and Uniform Resource Identifiers (URI) [33]. Fielding wanted to improve the HTTP without

breaking the Web [7]. The dissertation was done in parallel with Fielding authoring the Internet standards for the HTTP/1.1 and URI, which define generic the interface on the World Wide Web. Fielding is also one of the editors in developing Web standard for the World Wide Web Consortium's (W3C) "Do Not Track" and co-founder of the Apache Software Foundation. Fielding is a major contributor in the world of Web [7].

Fielding gives a following description of REST:
"The name "Representational State Transfer" is intended to evoke an image of how a well-designed Web application behaves: a network of web pages (a virtual state-machine), where the user progresses through the application by selecting links (state transitions), resulting in the next page (representing the next state of the application) being transferred to the user and rendered for their use." [32].

Figure 3.1 presents the state transitions of a Web page in state 1 transferring to state 2 after the link has been selected.

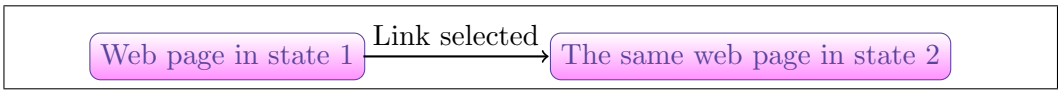

Figure 3.1: Representation of resource in state transition.

The Web consists of resources, which can be any information. For example, customer identified with an id = 1 can be presented as a resource, which is accessible by the following URL

http://www.builderscustomerportal.com/customer/1

A representation of the resource is returned, for example, customer1.html. The returned representation sets the client application in a state. When a hyperlink is selected from that page customer1.html, the new result will set the client into new representation state. Thus, the client application changes (transfers) it's state in each representation of the resource. That is why it is called Representational State Transfer [27].

3.2 Constraints of REST Architectural Style

The modern Web architecture emphasizes scalability, the generality of interfaces, independent deployment of components, minimizing latency, enforcing

security, and encapsulating legacy systems [33]. REST is a coordinated set of architectural constraints that attempt to minimize latency and network communication and maximize the independence and scalability of component [33]. To reflect the desired properties of a modern Web architecture REST defines the following six constraints (the last is optional) as part of an architectural style.

3.2.1 Client-Server

A client-server style is the most common web architectural style. A server component offers services to clients and is waiting for a request to be made upon these services. A client component requests a service from the server, which either rejects or performs the request and sends a respond back to the client. According to Andrews, [19] client-server system is a process, where the client is a triggering process and the server is a reactive process. The clients request triggers a server, which in return reacts by responding to the clients requests.

The Client-Server (CS) constraint is about separation of concerns, i.e., separating software components [47] via modularization. Modular programming emphasizes separating the functionality of a program into independent modules accessible through interfaces. In the Client-Server constraint, the interface (i.e. API) enforces separation of concerns in the system design [60]. The separation moves all the interface functionality into client-side. It simplifies the server component in order to improve scalability. The separation minimizes coupling, i.e., the degree of dependency between two elements. Furthermore, the separation allows both the client-side and server-side to evolve independently, requiring that the interface does not change.

3.2.2 Stateless

A communication between the client (server consumer) and the server (server provider) is stateless. More precisely put, the server-side component is stateless, thus, no session state is allowed on the server-side [32]. All of the application states is entirely kept on the client-side. Each client request is treated independently from the past or concurrent clients. They must contain all the necessary information and cannot use any stored context on the server.

The Client-Stateless-Server (CSS) constraint allows the application to scale [20]. In addition, it makes the server components to be lightweight. However, a disadvantage of client-stateless-server is that it may cause an overhead. The overhead is caused by increasing data repetition sent by requesting the same services causing a decrease in a network performance [32].

3.2.3 Cache

Client applications should cache responses, that are identified cacheable. [32]. Cacheable responses can be reused for later responses. Thus, the requests are equivalent to later requests and it will produce the same response. Client-Cache-Stateless-Server (C$SS) constraint reduces latency, will speed up the user-perceived performance and reduce a load on the network and server-side components [20].

3.2.4 Uniform Interface

All servers and clients within a RESTful architecture share a single common interface for all operations. The interface between clients and servers decouples components from each other [32]. A uniform interface constraint improves simplicity and visibility of the system by making it possible for both sides to evolve independently. However, the trade-off is a decrease in efficiency, due to transferring information in a standardized form.

The uniform interface constraint is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, Hypermedia As The Engine Of Application State (HATEOAS). These are the constraints related to Web API and will be discussed in more details in section 3.2.7.

3.2.5 Layered System

A REST compliant systems can be comprised of multiple architectural layers where no layer can "see past" the next [29]. A client cannot ordinarily tell whether it is connected directly to the end server or to an intermediary. Intermediary servers may improve system the scalability by load balancing and by providing shared caches. Layers can encapsulate legacy services and

protect new services from legacy clients. Each layer provides services to their neighbors, each layer having the limited view to their neighbors. This constraint makes it possible to evolve the system by adding, removing and changing layers without changing the client-side code. The trade-off is a reduction in performance [32].

3.2.6 Code-On-Demand

The server extends the functionality of the client by sending back code that the client needs to execute. Code-On-Demand (COD) typically relies on the use of Web-based technologies, such as JavaScript, where the client can download a javascript sent from a server [29].

The primary purpose of Code-On-Demand constraint is to allow logic within clients (such as Web browsers) to be updated independently from server-side logic. However, this reduces visibility, which is why this constraint is optional. Code-On-Demand is the only optional constraint, that is, architectures that do not use this feature are still considered RESTful [32].

The overall picture of REST can be seen in Figure 3.2.

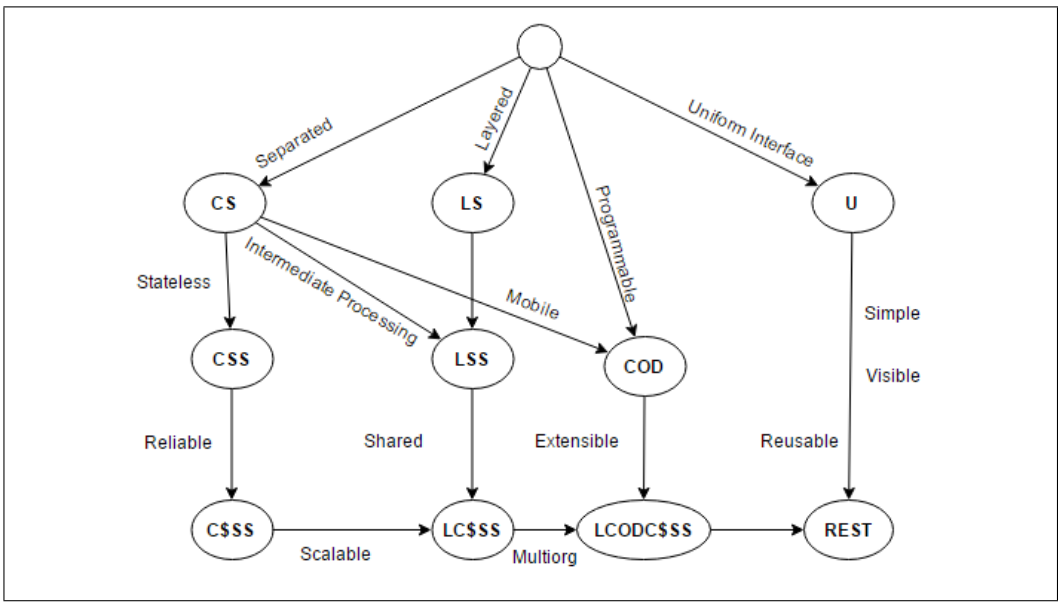

Figure 3.2: REST derivation by style constraints, adapted from [32].

3.2.7 Uniform Interface Constraint for Web APIs

The four constraints of the uniform interface are resource identifiers, resource representations, self-descriptive messages, and HATEOAS.

3.2.7.1 Resources and Resource Identifiers

REST is based on two keywords: resources and representations. A word resource is for information abstraction in REST [32]. The resource can be any information: a document, an image, a service (e.g. "today's weather' in Otaniemi'), a collection of resources and so on. The resource can be anything as a hypertext reference, which either perform on the resource or act as a new representation of the resource.

RESTful services based on HTTP use unique Uniform Resource Identifier (URI) for each resource [32]. For example the following URL

http://www.builderscustomerportal.com/customer/1

may display customer details with ID number one.

3.2.7.2 Media Types for Representations of Resources

Media types are ways to represent resource passed between client and server [1]. Every resource has at least one representation, which can be a document or image. The representations are returned in media types [32]. Most frequently adopted media types are JavaScript Object Notation (JSON) and eXtensible Markup Language (XML) [32]. This thesis uses JSON text format in a representation of resources.

JavaScript Object Notation (JSON) is a lightweight, text-based, language-independent text format for the serialization of structured data [9]. JSON can represent four primitive types and two structured types. The four primitive types are strings, numbers, booleans, and null and two structure types as objects and arrays. JSON is minimal, portable, textual, and a subset of JavaScript. An example of JSON presentation of a customer is in Figure 3.3.

```
{
    "CustomerID": "1",
    "FirstName": "John",
    "Lastname:": "Smith",
    "Age": "40",
    "Address" :
    {
        "StreetAddress": "100 MAIN ST PO BOX 1022",
        "City": "Seattle",
        "ZipCode": "981042,
        "StateCode": "WA"
    }
    "Phone": "123-456-7779"
}
```

Figure 3.3: Customer represented in JSON text format.

3.2.7.3 Self-Describing Messages

Each client request and server response with a self-descriptive message. That means each message contains all the information necessary for clients and servers to interact and complete the task. A resource's desired state can be represented in a client's request message and a resource's current state may be represented in the server's response message. Thus, a self-descriptive messages are stateless and context-free. The server may accept or deny the client's request in a response message. In HTTP, each message consists of headers and a body [1].

3.2.7.4 Hypermedia as the engine of application state (HATEOAS)

Hypermedia As The Engine Of Application State (HATEOAS) [32] is one of the main constraints of REST. Hypertext is text which contains links to other texts [65] and hypermedia is a term used for hypertext which is not constrained to be text. For example, hypermedia can include graphics, video, and sound. Some people also refer to HATEOAS simply as a hypermedia constraint [12].

A hypermedia-driven site provides information to navigate the site's REST interfaces dynamically by including hypermedia links with the responses.

HATEOAS definition according to Fielding [31]:
"The simultaneous presentation of information and controls such that the information becomes the affordance through which the user (or automaton)

obtains choices and selects actions."

In other words, a client can read pages and either follow links or submit forms.

Another way to approach HATEOAS is to think an application as a finite state machine, which consists of all the possible state of the application and the transition happens via hypermedia, that is given URLs in representations. HATEOAS is easy to implement for small and simple applications, but with something more complex the number of states may easily explode. HATEOAS may be complex to implement to the whole application. But can still be applied in a small and simple part of the application. The HATEOAS constraint decouples client and server by improving the independent evolvability. HATEOAS enhance discoverability, both humans and machines can follow links. With HATEOAS no versioning is needed [7]. If the entity needs to be removed or changed, only the URI is removed or changed respectively. REST architectural style was designed to model applications that evolve in the scale of decades [7] [32].

3.3 Summary

REST is an architectural style based on the design and architecture of the Web. It is used for distributed hypermedia such as World Wide Web. REST was designed to support software engineering over decades. Web APIs that adhere to REST style are called RESTful Web APIs. The next chapter shows how REST architectural style is applied to Web APIs.

Chapter 4

RESTful Web API

Web APIs that adhere to REST architectural style are called RESTful Web APIs. This chapter shows how RESTful style is applied on Web APIs in practice.

4.1 Building RESTful Web APIs by the REST Maturity Model (RMM)

Richardson REST Maturity Model (RMM) [61] is meant to guide a developer to build a RESTful API in four ascending levels. The higher an API adheres to the levels, the more mature it is in the RESTful style. RMM consists of 4 levels (0-3). Level 0 the tunneling style discussed in Section 2.3.1. It consists one Uniform Resource Identifier (URI) and one Hypertext Transfer Protocol (HTTP) verb. The first step to RESTful approach is adding a unique URI for each resource in level 1, which still has one HTTP verb. The next step in level 2 is to add HTTP verbs and final step in level 3 is adding Hypermedia As The Engine of Application State (HATEOAS). RMM with 4 maturity levels is shown in Figure 4.1.

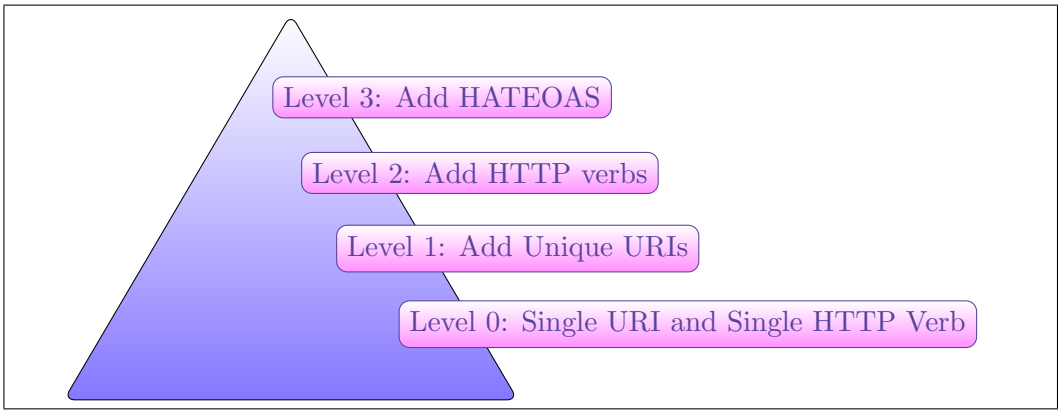

Figure 4.1: Diagram of the Richardson's REST Maturity Model adapted from [37].

4.1.1 Level 0: One URI and One HTTP Verb

In level 0 HTTP is used only as a tunneling system, where only one URI and one HTTP verb is used for different operations and resources. As discussed in Section 2.3.1, the tunneling style resembles the most RPC/SOAP services [46].

Consider the example of a customer submitting a service request using RPC/SOAP adapted from [20]. The system exposes a service at the URL /serviceRequest.

To submit service request, a client send the following:

```
POST /serviceRequest HTTP 1.1
Content-Type: application/xml
Content-Length: xx


<submitServiceRequest id = "1">
</submitServiceRequest>
```

The server responds that the service request is submitted:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: xx


<submitServiceRequestResponse>
Service request submitted
</submitServiceRequestResponse>
```

Note that instead of using an HTTP status code 201 Created the status is in respond body. HTTP is being used only as a transportation protocol. Another example of responding the submitted service request would be the following:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: xx


<submitServiceRequestResponse>
      <error code = "100">Submitting a service request failed<error >
      <reason>Missing information</reason>
</submitServiceRequestResponse>
```

Similar to the previous respond instead of using HTTP status code the error code is part of the respond body.

To check all the service request that are "In Warranty Request", the client sends a getServiceRequests request:

```
POST /serviceRequest HTTP 1.1
Content-Type: application/xml
Content-Length: xx


<getServiceRequest name= "In Warranty Request">
</getServiceRequest>
```

The server respond with the list:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: xx


<getServiceRequestResponse>
      <serviceRequests>
            <serviceRequest id = "1" name= "In Warranty Request"/>
            <serviceRequest id = "2" name= "In Warranty Request"/>
      <serviceRequests>
</getServiceRequestResponse>
```

4.1.2 Level 1: Unique URIs for Resources

In level 1 the first step to more RESTful Web API is to propose resources [46]. Level 1 provides unique URIs for different resources. A resource is any information that is made available for clients. The idea is to design URIs analogically with the resource set. Take, for example:

All the customers:
http://www.homebuilder.com/api/customers

A customer with id=1:
http://www.homebuilder.com/api/customers/1

To create a customer with identity id = 1:

```
POST http://www.homebuilder.com/api/customers HTTP 1.1
Content-Type: application/json
Content-Length: xx


{
      "id" = "1",
      "FirstName" = "John",
      "LastName" = "Smith"
}
```

The server responds that the customer is created:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: xx


<createCustomerResponse>
A customer with id = 1 created
</createCustomerResponse>
```

Notice that in Level 1, the status of response is still part of the body response.

To retrieve the customer with identity id = 1:

```
POST http://www.homebuilder.com/api/customers/1 HTTP 1.1
Content-Type: application/json
Content-Length: xx
```

The server responds with the JSON presentation of the customer:

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Length: xx


{
      “id” = ”1”,
      “FirstName” = “John”,
      “LastName” = “Smith”
}
```

However, Level 1 still uses only one single HTTP verb like POST for retrieving the resources.

4.1.3 Level 2: HTTP Verbs

According to Richardson's REST Maturity Model [61] shown in Figure 4.1, an API should be using HTTP verbs, in order to be more RESTful. Note that REST is an architectural style and is completely protocol agnostic. REST does not force to use HTTP protocol [32]. This thesis applies RESTful Web APIs over HTTP.

In level 1 POST is used for all methods, but in level 2, the idea is to map HTTP verbs as consistent as possible, e.g., GET for retrieving, POST for creating a resource and DELETE for removing a resource.

Adding different HTTP verbs limits how to interact with resources. Clients operate on resources with the following HTTP verbs: GET, POST, PUT, and DELETE. These are the main verbs and define the uniform interface [34]. GET-verb signifies that the resource is fetched as read-only data. POST indicates that a new resource is created. PUT is used for updating the existing resource. DELETE indicates that the client wants to delete the resource. See figure 4.2. These operation types are called also CRUD (Create, Retrieve, Update, and Delete).

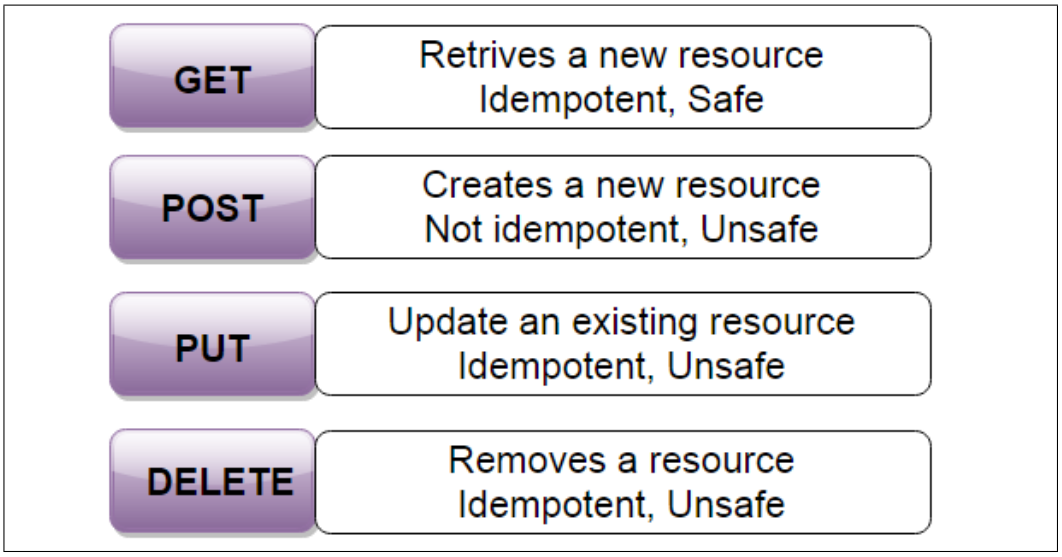

Figure 4.2: Uniform Interface, adabated from [34]

In level 2 an API should be able to deal with caching, scalability and failures [46]. Excluding POST, another benefit is that the rest of the HTTP verbs: GET, PUT, and DELETE are idempotent [34]. Idempotent means

that a certain action can be applied multiple times without changing the result.

GET on a service can be called multiple times without a change in the resource. The same update can be PUT with the same result. Using DELETE on the same resource again is not possible. The only unsafe verb is POST. More detailed information about POSTing a resource in a safe environment is described in subsection 6.5.

GET is a safe method. Safe methods do not modify resources. Thus, safe methods can be cached. Caching is important in order to service to be scalable [32]. Idempotency is important in building a fault-tolerant API [20]. Take for example POST, which is not an idempotent verb, calling it multiple times creates multiple resources. For example, what happens, if a POST request is sent out to the server and the request times out. Is the resource actually created? Is it safe to retry again? With idempotent methods, it is safe to resend the request until getting a response from the server. Non-safe (and non-idempotent) methods will never be cached by any middleware proxies [20].

4.1.3.1 HTTP status codes

Clients request a service from a server and the server response to client's request by the 3-digit integer HTTP status code [65]. It is a bad practice to use HTTP status code 200 (OK) when something is wrong or only HTTP status code 500 (Internal Server Error) for all the bad client's requests. The 2xx class of status codes indicates success in client's request. The 4xx class of status code implies client error and the 5xx class means server error in the request. More details about HTTP status codes will be found from [65]. The meaning of HTTP status codes is described in Appendix A.2

In Figure 4.3 a client requests a customer resource with GET HTTP method and a server responds with HTTP status code 200 OK appended with the customer as JSON media type format.

4.1.4 Level 3: HATEOAS

Level 3 Hypermedia As The Engine of Application State (HATEOAS) is one of a precondition for RESTful applications. The principle behind HATEOAS

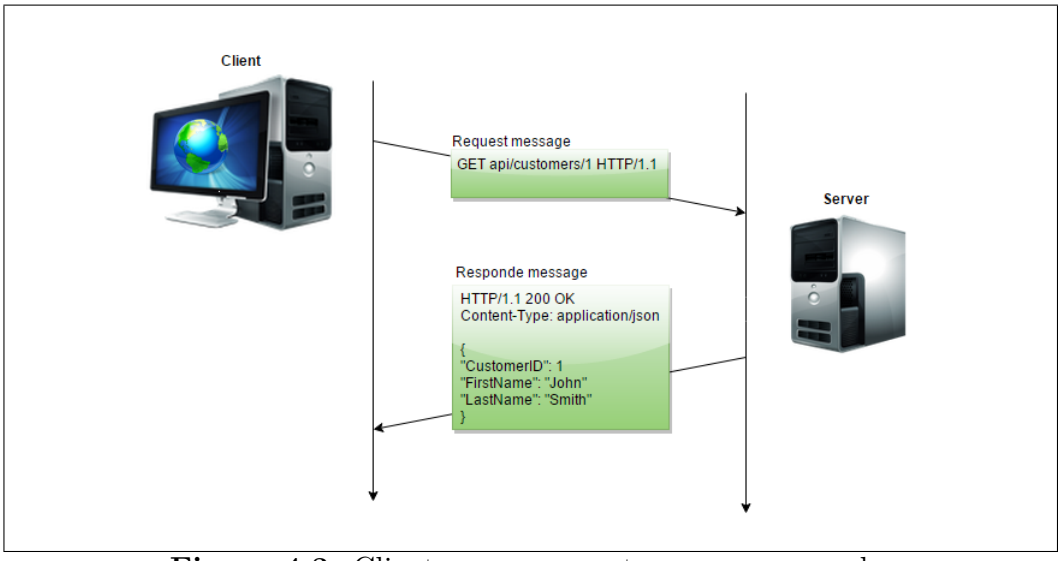

Figure 4.3: Client-server request-response example.

is that a Web application can navigate from one state of the application to the next state entirely through hypermedia links provided dynamically by an application server [46]. A REST client needs no prior knowledge about what actions to take next. The World Wide Web is the best example for hypermedia. The user starts from a home page and navigates following different links to another representation of the web page. Similarly, in REST compliant applications, a client makes transitions through an applications state only by navigating through hypermedia links provided within a resource representation returned from the server. That is, given a starting URI, the client should be able to navigate without prior knowledge of the possible navigation paths. Whenever a resource is returned from the service, it should include the URIs that can be applied in the next request.

The RMM provides a good model for developing RESTful Web API, however, it is not a definition of constraints itself [46]. If the application adheres only to a maximum of level 2 it is not considered RESTful. Applications that are on level 3 are considered to be RESTful. However, even that still does not mean that the whole service is RESTful. Roy Fielding stated that level 3 HATEOAS of RMM is a precondition of REST.

"What needs to be done to make the REST architectural style clear on the notion that hypertext is a constraint? In other words, if the engine of application state (and hence the API) is not being driven by hypertext, then it cannot be RESTful and cannot be a REST API. Period. Is there some

broken manual somewhere that needs to be fixed?" -Roy Fielding [31]

For example, the following HATEOAS JSON presentation for a customer with status "Prospect". The customer is a prospective home buyer who can print saved brochures.


```
{
    "CustomerID": "1",
    "FirstName": "John",
    "Lastname:": "Smith",
    "Status": "Prospect",
    "links": [
         {
        "rel": "self",
        "href": "http://homebuilders.com/api/customer/1",
        "verb": "GET"
         },
        "rel": "printBrochure",
        "href": "http://homebuilders.com/api/Lead/PrintBrochure/2",
        "verb": "GET"
         }
    ]
}
```


When a prospective customer decides to buy a house and becomes a homeowner the status of the customer is updated respectively as "HomeOwner". Now in addition to printing brochures the homeowner can also view its home information and submit service request related to the house as following HATEOAS JSON presentation suggest.


```
{
    "CustomerID": "1",
    "FirstName": "John",
    "Lastname:": "Smith",
    "Status": "HomeOwner",
    "links": [
         {
        "rel": "self",
        "href": "http://homebuilders.com/api/customer/1",
        "verb": "GET"
         },
        "rel": "Print brochure",
        "href": "http://homebuilders.com/api/Lead/PrintBrochure/2",
        "verb": "GET"
         },
        "rel": "Home info",
        "href": "http://homebuilders.com/api/Home/GetInfo/3",
        "verb": "GET"
         },
        "rel": "Submit service request",
        "href": "http://homebuilders.com/api/Home/SubmitServiceRequests/",
        "verb": "POST"
        }
    ]
}
```

4.2 Summary

This chapters provided a detailed description of RESTful Web APIs. The next two chapters discuss versioning and security of Web APIs for a customer portal.

Chapter 5

Versioning

Business concerns are changing in time and APIs must evolve to meet the changing needs. The question is, how to create a Web API that can evolve over a period of years. There are two choices; either to go along with the existing API adding new features without breaking anything or create a new one [20]. If a change in an API may break a contract, that is an application built on it, the breaking change should be introduced in a next version.

Up to this chapter, this thesis has discussed Web API, REST style applied on HTTP protocol and how to build a RESTful Web API by 4 steps of RMM. The next two chapters discuss versioning and security.

This chapter focuses on versioning of Web API. Where possible, the constraints of the REST architectural style will be applied on versioning. REST is not an end goal, but rather a means to reach the goal. It is possible that some REST constraints are violated in order to achieve evolvability in Web API.

5.1 When Is Versioning Needed

If there is a breaking change in a Web API contract, it should be introduced in a next version. The next question is what is the contract in a Web API? In a Web API, clients have access to representations of resources via URIs. From this point of view, the contract in Web APIs consists of

1. Resource
2. Representation of resource
3. URI as a Resource identifier

URIs and representation of resource are used to build a web application on top of its API. In the next sections, different breaking change in JSON representation and URIs are introduced.

5.1.1 Case 1: Adding New Fields to Representation

Customer JSON:

```
{
    "CustomerID": "1",
    "FirstName": "John",
    "Lastname:": "Smith",
    "Age": "40",
    "Address" :
    {
        "StreetAddress": "100 MAIN ST PO BOX 1022",
        "City": "Seattle",
        "ZipCode": "981042,
        "StateCode": "WA"
    }
    "Phone": "123-456-7779"
}
```

In case 1 adding new fields to representation is not a breaking change, for example, a new data field, e.g., “Fax” is added to a customer JSON as follows

```
{
    "CustomerID": "1",
    "FirstName": "John",
    "Lastname:": "Smith",
    "Age": "40",
    "Address" :
    {
        "StreetAddress": "100 MAIN ST PO BOX 1022",
        "City": "Seattle",
        "ZipCode": "981042,
        "StateCode": "WA"
    }
    "Phone": "123-456-7779"
    "Fax": "123-456-7778"
}
```

Adding contents to a representation does not break the API contract, clients will not be aware of something that is not used before. They will get the same presentation with additional fields. Hence, no version is needed.

5.1.2 Case 2: Renaming or Removing Fields from Representation

Renaming or removing fields is a breaking change and in such cases the version is used. For example renaming data field “Phone” to “PhoneWork” would break the application build on top that API, because references to the customer.Phone data field would not exist anymore.

5.1.3 Case 3: Changing Resource Entity

Changing a resource entity entirely is a breaking change. Instead of versioning it is more reasonable to introduce a new Web API, a new resource with its own URI.

5.1.4 Case 4: Changing URI

Changing a URI is a breaking change. The URI is a unique resource identifier to resource. From the RESTful point of view, changing URI means it points to an entirely new resource entity. Practically, it is the same as changing a resource entity.

5.1.5 Case 5: Change in a program running under API

If a change in a program does not violate the API contract, then there is no breaking change.

5.2 Versioning methods

Versioning can be made several ways. Two most common ways to make API versions; the URL way and the hypermedia way [44].

5.2.1 The URI Way

The URI way is a simple way for versioning an API. This versioning is specified the a ‘v‘ prefix. And is placed all the way in the left so that it has the highest scope.

Version 1: /api/v1/customer/1

To update to the next API version, increase the version number

Version 2: /api/v2/customer/1

Only positive integers are used.

The advantage of versioning the URI way is that clients know which version they are using. The drawback is that it is not RESTful because different version URIs still link to same resource entity [7]. Another drawback is that the older API versions need to be maintained also and this may cause overhead in a Web API development [30].

5.2.2 The Hypermedia Way

The hypermedia way in accept headers, which describes how the data is represented, for example below:

```
GET /api/customer/1 HTTP/1.1
Accept: application/json; version=1
```

This technique versions the representation. However, it is a more complicated approach and it is also more difficult to test. Instead of introducing new version number in a URL with few characters, a developer has to add a relatively high number of line of codes to address the right version of a representation of a resource [13]. Clients cannot test straightforwardly via a URI. Instead, they have to construct a request and configure an appropriate accept header [15].

From these two ways, the URI versioning is for both client and server-side developer easier to consume.

5.3 Summary

This Chapter introduced different version cases and how they should be versioned. Next chapter discusses the security of Web APIs. How Web APIs are authenticated and authorized? and ends with security analyze of data posted to a server.

Chapter 6

Security

Previously, this thesis has discussed how to create a Restful Web API and versioning of a Web API. Security is also an important part of the design. Web API security is a major concern when requesting resources via URIs. Without security, the resources are available to everyone to operate on. Web API security is based on two questions: first, is the user of the resource really who he claims to be and does that user have access to that resource [46]. This chapter will cover the security of a Web API and introduce a token based authentication used in a customer portal. Then it will discuss a validation of unsafe POST verb.

6.1 Authentication and Authorization

Authentication validates user identity with a question, "Is the user of the API service who he claims to be?" [46] For example, a user logs in with his username and password and the server authenticates the user by his password.

Authorization is challenged the user's permission with a question, "Is the user allowed to do what he is trying to do?" [46] For example, a user is allowed to get a resource but not to create a resource.

6.2 Token Based Authentication

Before token based authentication, the traditional way to have application to remember user logged in is store user info in a session. On every client's request, the server checks the session state and then responds. Session state on the server can cause overhead when there are many users authenticated. Another problem is since sessions are stored in the server memory, it does not allow the server to scale [32].

Token-based authentication is stateless. No information about the user is stored on the server or in a session [65]. Stateless means that if a client authenticates a user with a username and password, then on the next client's request, the server won't know who is requesting the resource. The client would have to authenticate again. Stateless allows the application to scale. Furthermore, token based authentication decouples the client and the server, since authentication and authorization are handled only in the server side.

The basic steps in the token-based authentication are as follows:

1. A user logs in with a username and a password.
2. An authorization server validates credentials.
3. The authorization server provides a token to the client.
4. The client stores the token and sends it along with every user's further request.
5. The server verifies token and responds with a representation of a resource.

Figure 6.1 depicts the token based authentication steps.

The token is used for authentication and authorization of the user. Stateless tokens that are valid forever will be a problem. The token should consist an expiration time. In this thesis the token consist of three parts; userID, expiration time and a Hash-based message authentication code (HMAC). HMAC is used to validate data integrity and authentication of the token's userID and an expiration time of the token.

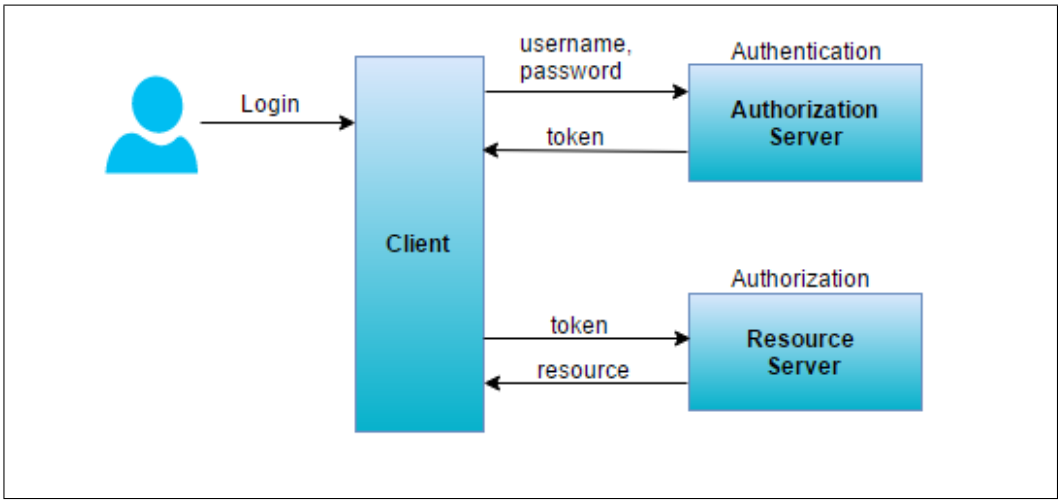

Figure 6.1: A token credentials flow, adapted from [3].

Authentication of a message means that a message has not been modified (data integrity) or changed while in transit and the receiving third party can verify the message. Data integrity detects accidental and intentional message changes while authenticity verifies the origin of the message [21].

6.3 Hash function

A hash function **H** is an algorithm that takes a variable-length of input data **m** and generates a fixed length string, which is called the hash value **h** [21]. Message authentication code (MAC) is a particular use of the hash to verify the integrity of the data received.

Hashing is used to secure the communication since hash algorithms are fast and energy efficient among cryptographic algorithms [21] [58]. Hash $\mathbf{h_1}$ is generated from credentials **m** and is concatenated with credentials (denoted by |) $\mathbf{m} \mid \mathbf{h_1}$ as token. When authorizing the token, a new hash $\mathbf{h_2}$ is calculated from the credentials given in the token. If $\mathbf{h_1}$ equals $\mathbf{h_2}$, the authentication is successful.

A cryptographic hash function is a function which is considered practically impossible to invert, that is, one cannot recreate the input data from hash value alone [21]. Hash functions with only the property of compressing a message to smaller size have diverse computational uses, but when used in cryptography the hash algorithm needs more properties.

The basic requirements of a cryptographic hash function are listed below [21]:

- H(x) is relatively easy to compute the fixed length hash value h for any variable length message x
- H(x) is one-way mapping:
  - it is infeasible to generate a message from its hash
  - it is infeasible to modify a message without changing the hash
- H(x) is strongly collision-free:
  - it is infeasible to find two different messages x, y with the same hash H(x)=H(y)

Hash functions are usually designed from scratch or made out of a block cipher in a black box way [25]. Hash functions based on block ciphers is illustrated in Figure 6.3. Some of the known hash functions constructed from scratch are SHA-family [53] and MD5 [62].

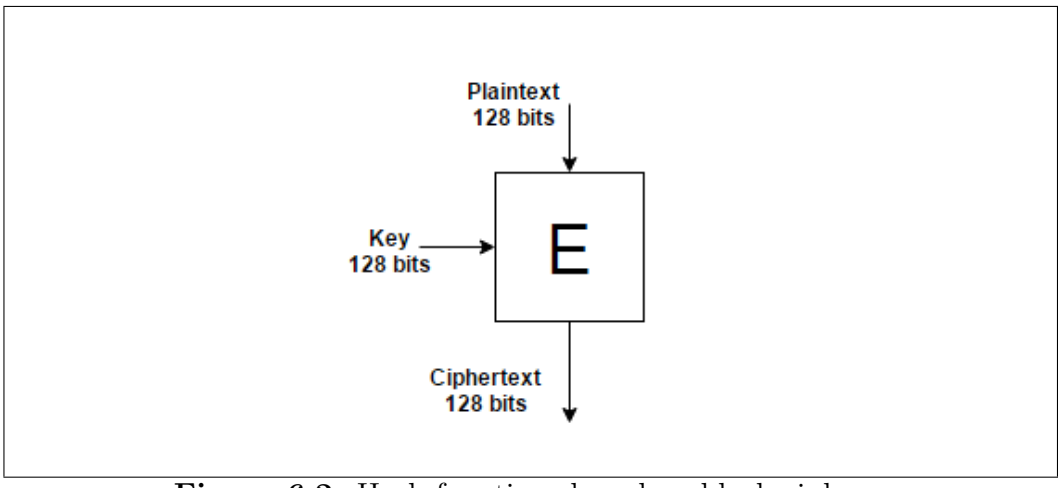

Figure 6.2: Hash functions based on block ciphers.

6.3.1 SHA

The Secure Hash Algorithm (SHA) [28] is a family of cryptographic hash function published by the National Institute of Standards and Technology (NIST) as a U.S. Federal Information Processing Standard (FIPS) [53] including SHA-0, SHA-1, SHA-2 and SHA-3 families.

This thesis uses SHA-256 from the SHA-2 family hash functions. SHA-256 produces a fixed length hash value of 256 bits or 32 bytes. SHA-256 is cryptographically stronger hash function than MD5 (hash length of 128 bits or 16 bytes) and SHA-1 (hash length 160 bits or 20 bytes) [26] [21].

What is achieved in security will be lost in performance, that is, MD5 is fastest, then SHA1 and the SHA256 is the slowest from the three [21]. It is not recommended to use MD5 and SHA1 since they have known security vulnerabilities [21] and they are already broken in several studies [38] [66]. Although, SHA-1, till date, is still the most widely used hash function, in spite of several successful cryptanalytic attacks against it [38]. However, the attacks remain impractical due to high computation complexity and associated cost [38]. There is no known SHA-256 break yet.

6.4 HMAC

Message authentication is a prime necessity in the world of open computing and communications. Hash-based message authentication code (HMAC) algorithm's purpose is to verify the integrity and the consistency of the data [45]. Mechanisms that provide such an integrity check based on a secret key are called "message authentication codes" (MAC). Typically, message authentication codes are used between two parties that share a secret key in order to validate a message transmitted between these parties.

HMAC uses a secret key in conjunction with a hash function to produce a hash that is appended to the message. Any cryptographic hash function, such as Message Digest 5 (MD5) or Secure Hash Algorithm (SHA-1), may be used in the calculation of an HMAC: the resulting MAC algorithm is termed HMAC-MD5 and HMAC-SHA1 accordingly [45]. Security strength of the HMAC depends on the cryptographic strength of the underlying hash functions, the size of its hash output, and on the size and quality of the key.

An iterative hash function breaks up a message into blocks of a fixed size and iterates over them with a compression function. For example, MD5 and SHA-1 operate on 512-bit blocks. The size of the output of HMAC is the same as that of the underlying hash function (128 or 160 bits in the case of MD5 or SHA-1, respectively), although it can be truncated if desired.

HMAC definition from [45]

$$HMAC(K, m) = H((K \oplus opad)|H((K \oplus ipad)|m)) \tag{6.1}$$

where, H is a cryptographic hash function
K is a secret key
| denotes concatenation
$\oplus$ denotes exclusive or (XOR)
opad is the outer padding (0x5c5c5c...5c5c, hexadecimal constant)
ipad is the inner padding (0x363636...3636, hexadecimal constant)

K is a secret key padded to the right with extra zeroes to the input block size of the hash function. If the secret key is longer than that block size, it is a hash of the original key.

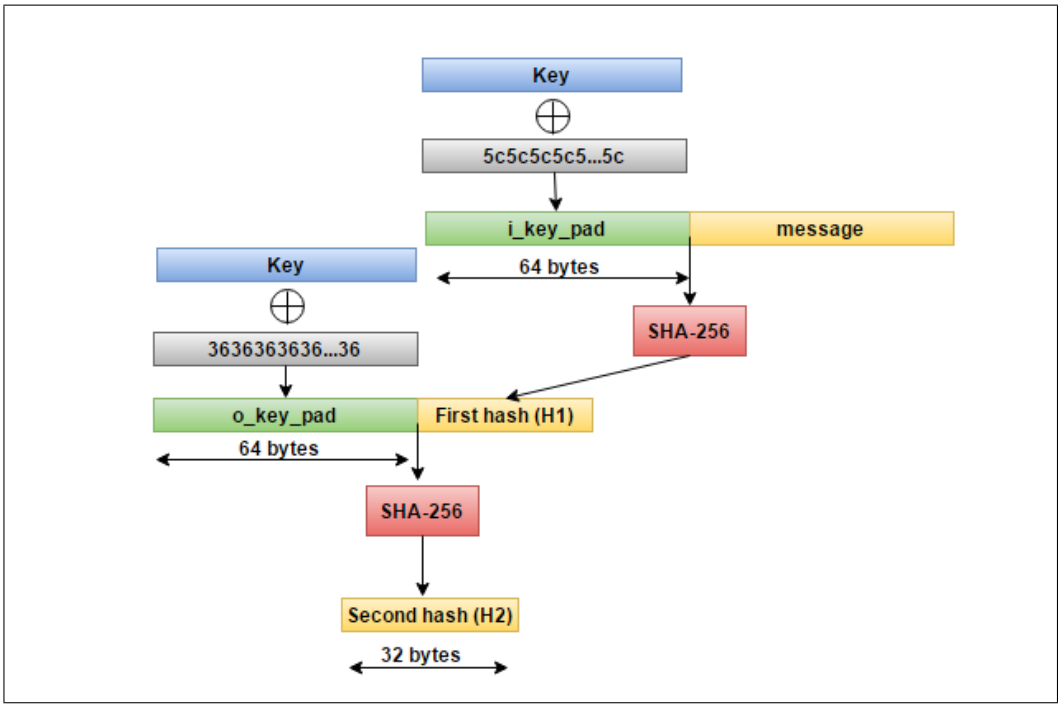

Figure 6.3: HMAC-SHA256 generation.

The token based authentication is using HMAC-SHA256 to authenticate the userID and expiration time of the token.

6.5 Unsafe and Non-idempotent POST

This thesis uses ASP.NET Web API as a framework for developing Web APIs. In this framework, HTTP verb POST is used for both creating and updating a resource. The main HTTP methods used in this thesis are GET and POST. Therefore, this thesis will only focus on the security analysis of GET and POST. As discussed in Section 4.1.3 GET is both idempotent and safe verb, POST in the other hand is not idempotent and not safe. A safe verb does not modify a resource and an idempotent verb can be called many times without different outcome. By definition, safe verbs are also idempotent verbs.

Usually when saving or updating received data sent by a client, it is validated before doing any further processing. The main goal is that instead of corrupt data only valid data is saved into the server. This section discusses data annotation in validation, under and over posting problems.

6.5.1 Data Annotations

ASP.NET Web API uses attributes to set validation rules for properties [68]. Consider the following model of a customer, which represents the homeowner with HomeID:

```
public class Customer
{
    [Required]      public string HomeID        { get; set; }
    [Required]      public string Name          { get; set; }
    [Range(0, 100)] public int Age              { get; set; }
                    public string StreetAddress { get; set; }
}
```

The “Required” attribute says that the properties “HomeID” and “Name” must not be null and the “Range” attribute determines that the property “Age” must be between 0 and 100

Suppose that a client sends a POST request with the following JSON representation:

```
{
    "HomeID": "1",
    "Age": "45",
    "StreetAddress":  "16158 Redmond Way Ste 188"
}
```

Note that the client did not include the required "Name" property. When Web API converts the JSON into a Customer instance, it validates the validation attributes. During validation, it cannot find the "Name" property. Thus, this model state is not valid.

6.5.2 Under Posting

```
public class Customer
{
    [Required]      public string HomeID        { get; set; }
    [Required]      public string Name          { get; set; }
    [Range(0, 100)] public int Age              { get; set; }
                    public string StreetAddress { get; set; }
}
```

Under-posting happens when some properties are left out [68]. For example, suppose that the client is sending the following:

```
{
    "HomeID": "1",
    "Name": "John Walker",
    "StreetAddress":  "16158 Redmond Way Ste 188"
}
```

The "Age" property is now missing. In ASP.NET Web API, the model binding is assigned a default value of zero for missing properties. Thus, the model state is valid. This might seem a desirable property in a POST creation but in a POST update it might cause the problem. In the POST update operation, it is good to have the choice to distinguish between "zero" and "not set" by adding "?"-character after type variable as follows. attribute [68]:

```
[Range(0, 100)] public int? Age { get; set; }
```

To force clients to set a value, make the property nullable by "?" character and set the Required [68]:

```
[Required]
[Range(0, 100)] public int? Age { get; set; }
```

6.5.3 Over Posting

A client can also include additional properties into a JSON [68]. For example:

```
{
    "HomeID": "1",
    "Name": "John Walker",
    "Age": "45",
    "StreetAddress":  "16158 Redmond Way Ste 188",
    "Gender": "Man"
}
```

A property "Gender" is included in the JSON and does not exist in the Customer model. In this case, the value is ignored. However, Over-posting is a problem if the model consists read-only properties that are not supposed to be modified. For example:

```
public class Customer
{
    [Required]      public string HomeID        { get; set; }
    [Required]      public string Name          { get; set; }
    [Range(0, 100)] public int Age              { get; set; }
                    public string StreetAddress { get; set; }
                    public bool IsAdmin         { get; set; }  // read only
}
```

In this case, an infiltrator might add "IsAdmin" = "true" and promote themselves to the administrator. The safe workaround for this is to introduce a model, which consist exactly the updateable properties.

```
public class APICustomer
{
    [Required]      public string HomeID { get; set; }
    [Required]      public string Name { get; set; }
    [Range(0, 100)] public int Age{ get; set; }
                    public string StreetAddress { get; set; }
}
```

6.6 Summary

This chapter discussed the security of Web APIs. The token-based message authentication HMAC-SHA256 of Web APIs was introduced. In addition, data validation in over-posting and under-posting is reviewed. Next chapter gives a background to the home building industry and a brief overview of Sapphire application and customer portal for home builders.

Chapter 7

Application

This chapter discusses home building industry in the United States of America. Furthermore, the chapter introduces SapphireBuild portal and describes different customer portal users.

7.1 The home Building Industry in the USA

The home building industry in the USA compasses several of the nation's largest publicly-traded home builders [48]. Strategically, all home builders follow a similar operating model, which is primarily purchasing a land and construction on that land. The home building market can be separated in three categories. Some home builders are in an affluent market providing luxury homes. Others focus on an entry-level category, where home buyers make their purchase decision based on their ability to secure affordable financing. However, a majority of home builders are in a "first-time buyer" segment. The price is slightly higher than that of an entry model.

At the last peak of the housing cycle in 2006 (see Figure 7.1), the ten biggest home builders constitute for about 35% of housing starts [48]. The majority of home builders consist of small private home builders. In good economic times, it is common for big home building companies to expand their land

positions by buying small regional home builders.

The home building industry accounts for 66bn in revenue with an annual growth of 3.7%. The industry employs 398,391 people over 251,773 businesses [41].

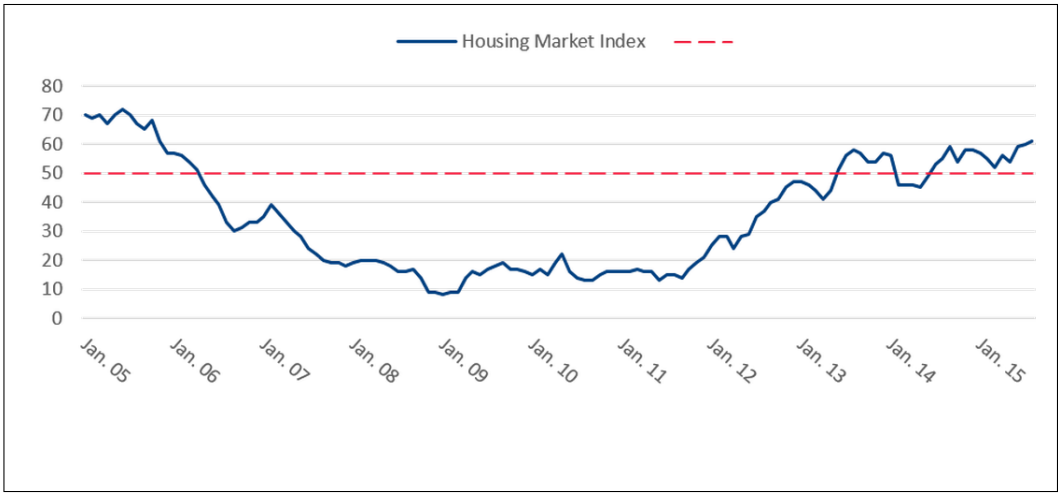

Figure 7.1: Housing market index [50].

Figure 7.1 shows that in last three years the home building industry of United State of America is recovering well from the last bottom of the housing cycle in 2009. In 2005 2 million new homes were started, in 2009 the number of started homes dropped to 554 000 and in 2014 is has increased to 1 million [22].

7.1.1 Types of Home Builders

Home building industry consists of three kinds of home builders: production builders, building manufacturers and custom builders [40]. These builders can be measured by the number of houses they build per year. The largest group is production builders, the second largest is building manufacturers and the smallest is custom builders. However, custom builder constitutes the majority of home builder companies in the housing business [40].

7.1.1.1 Production Builders

Production builders provide houses in communities. They own the land their houses are built on and usually participate in community development. The

houses can be either single-family or multifamily homes such as townhouses, apartment buildings, and condos [51]. Some production home builders provide houses in either of the categories and some in both. The production homes are pre-designed home models and can have design options, for example, to replace the bathtub with a larger shower. However, the changes are not possible to the extent with custom homes [40].

7.1.1.2 Building Manufacturers

Building manufacturers build their houses in factories. There are two kinds of home builders; mobile home builders and modular home builders [40]. Mobile homes are basically, houses on wheels which can be moved wherever the home buyer desires. Modular homes are prefabricated buildings built in multiple sections called modules. Modular homes are usually 78-80 percent complete before transported to a building site and then assembled. Unlike mobile homes, modular homes do not have axles or frame, therefore they are typically transported to their new location by flat-bed trucks [35].

7.1.1.3 Custom Builders

Custom builders typically create a unique home with the wide range of design options. Custom homes are attractive for home buyers that desire to select more details to their home. Home buyers can customize their home, which is generally not possible to such an extent with production builders and building manufacturers [54]. A custom home is more expensive than a production model of similar size and floor plan since high volume production builders save in buying materials in bulk.

This thesis focuses in production builders building single family homes. In the next two subsections, the Community Life Cycle and Sales Process will be described as it applies to production builders building single family homes. Details about building manufacturers and custom builders will not be discussed.

7.1.2 Community Life Cycle

Production builders construct houses in communities [54]. First the builder buys a piece of land to build the community on. After that, the builder

applies for the building permits and hires the contractors to build the community infrastructure. The community is divided into sections called lots. Builder builds showpieces of their models on some of the lots. One of them is used as a community sales office and the remaining lots are available for home buyers. From the sales and marketing point of view, the community life cycle ends when all the lots and also the model homes are sold out [40] [6].

7.1.3 Sales Process from Lead to Home Owner

A process starts when a prospective home buyer visits the home builder website, clicks a brochure of a home model and leaves a contact information [5]. From that moment on, they are considered as a lead, which is a possible prospect. Customers who fill in their contact information other way are also considered leads [39].

A lead manager scans through the leads records and will assign prospective home buyers to a sales representative. If there has been more communication between a sales representative and a lead, the sales representative can upgrade the lead to a prospective buyer by status "Prospect" [5]. On the other hand, leads, who visit the sales office and leave their contact information are also upgraded to prospects [5].

A prospective home buyer visits the sales office, where the home buyer is given a tour in the model homes. The prospect is shown possible options to select style or design from a menu in several product categories (such as flooring, appliances, and countertops). After signing the contract of buying a new home and paying earnest money, the new home buyer is allowed 30-60 days to visit the design center and to finish the rest of the selections [40]. During this time, home buyers under the sales contract are waiting for their house to be build. Home buyers become homeowners when they move into their new house [5]. Figure 7.2 illustrates the whole process from a lead to the homeowner.

7.2 Sapphire Build

A software company Kova Solutions Inc provides a software business solution SapphireBuild for local home builders in the USA [10]. The primary

platform is .NET, Microsoft Windows Server, and Microsoft SQL Server. KovaSapphire is a web-based operational management software for production builders to aid all stakeholders in the building process [17]. KovaSapphire lets different stakeholders work from the same database [24] with support for a complete home life cycle. This includes marketing, sales, home building process and customer relations management.

Homeowners can submit service requests on the Internet via a customer portal. Working from the same database allows both portals: a customer portal and SapphireBuild to have an access to a complete history of a home, including vendors and warranties. Thus, a Customer Service Team (CST) can effectively communicate with the homeowners via Sapphire Build [11]. Figure 7.3 depicts the communication via portals between a customer and a customer service team.

7.3 Customer Portal

A Web portal is a gateway allowing users to access diverse information and content found on the Internet [67]. In essence, a web portal is a website where divergent and aggregated data can be accessed from one single entry point [14]. A portal is a Web site or a Web service that provides information content to serve a specific audience.

A Customer Portal is a gateway for customers, vendors, and home building providers to distribute shared data [11]. Customers can log in, review and update their private information about their home and contact details as well as make service requests. The customer portal will provide an opportunity for customers for self-managing contact and home information. Homeowners can be in contact with any concerns 24/7. Direct communications between customers and customer services, through portals, may prevent costly mistakes before they happen. Customers can make a service request and follow the process. This will provide service transparency, access, and control to the service request process.

7.3.1 Different Portal Users

A customer portal is designed for different users; lead, prospect, a customer under contract and homeowner. The sales process from lead to homeowner was described in detail in Subsection 7.1.3.

7.3.1.1 Lead

A lead is an individual who has provided contact information [39]. For example by registering on a home provider site or printing a brochure of a home model for the first time, customers are asked to leave their contact information via web forms citeFromLeadToHomeOwner.

7.3.1.2 Prospect

A prospect is a prospective home buyer. Prospect can be either a customer who visits the sales office and provides contact information [5] or someone who has been in a two-way interaction with a lead manager [39] and being promoted to a prospect by the lead manager [5].

7.3.1.3 Under Contract

A prospective home buyer becomes a customer, who is under a contract after signing of a contract [5]. Home buyers under the contract are in a state of waiting for their house to be built.

7.3.1.4 Homeowner

Home buyers become homeowners when they get keys to their new home [5].

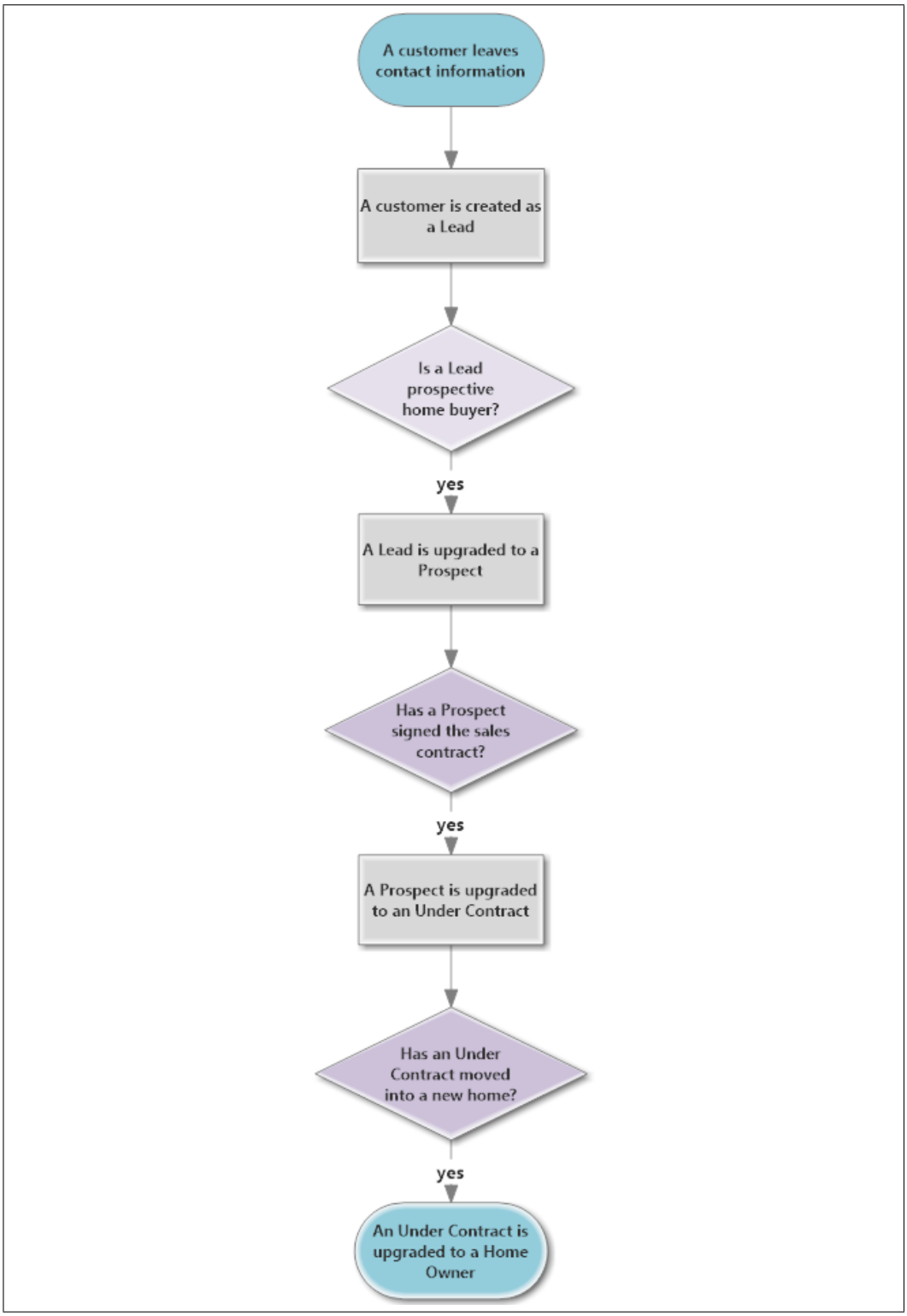

Figure 7.2: Sales process from Lead to Home Owner.

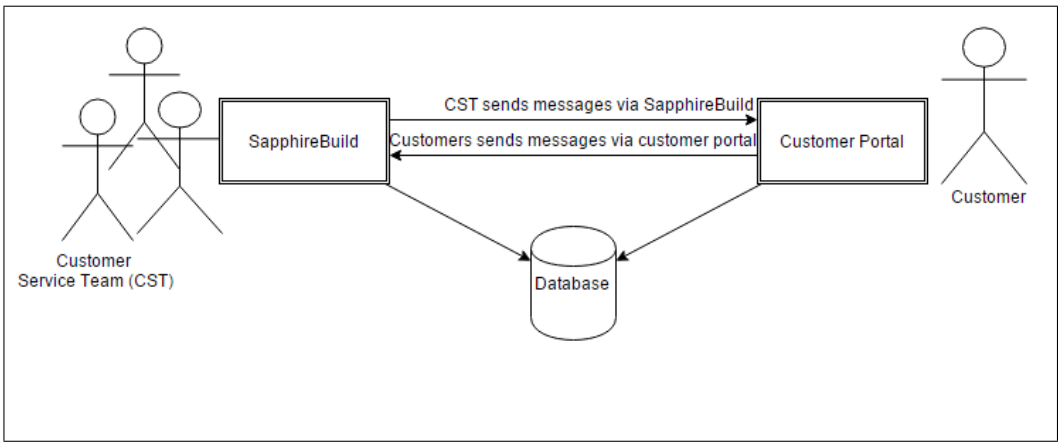

Figure 7.3: SapphireBuild and a customer portal operate from one database.

Chapter 8

Design and Implementation

This chapter covers the requirements, design, and implementations of Web APIs for a customer portal. IEEE 1471 [16] is adopted as the conceptual framework of the design. First the stakeholders of customer portal Web APIs are identified, after that their concern and needs are described.

8.1 Requirements

Stakeholders may be individuals, teams, or organizations with interests or concerns related to a system. The main users of a customer portal are different portal users such as leads, prospects, customers who are under sales contract and homeowners. A web-based customer portal gives 24/7 access to users. Users can view their home information, contact details and request services related to warranty. The customer portal will be role-based and the view will be customized according to the role of the portal user. The further requirements regarding each portal user will be described later in this section.

Web APIs for a customer portal are implemented and maintained by the developers. Web developers use Web APIs to implement and maintain their web-based customer portal. A Web API improves loose coupling and lets

both client and server side evolve independently. Developers can focus on providing Web API services and Web developers can focus on web designing.

A company that offers an API is called an API provider. People that use an API to create the applications are called developers. People that use an API based application are called end users [42].

Owners of the customer portals are home builders. They offer customer service via the customer portal. The customer portal is role based and is customized for each role of a user. Since this thesis focuses on the customer portal design, the requirements are described only for portal users. In the next section, the requirements for different portal users will be described.

Different users of a customer portal are Lead, Prospect, Under Contract, and Homeowner.

8.1.1 General

Each user, a lead, a prospect, a customer under contract and a homeowner in a customer portal log into their account using an email address as a username.

In a case the password is forgotten, a user can reset his or her password by providing a portal account related email. The reset link is then sent to the user's email. Via the link the user is asked to provide a new password.

In the portal user can change their password. The old password, a new password and a confirmation for the new password are asked in a changing password process. The password has some complexity requirements.

1. Login to the portal

2. Reset Password

3. Change Password

8.1.2 Lead

When a customer registers on the home builders site or prints a brochure of a home model for the first time, a new lead is asked to provide contact information. The lead is sent an email message containing a username, a password, and a link to the customer portal. Lead can view and reprint brochures in the customer portal.

1. Reprint their brochures

8.1.3 Prospect

After providing contact information via the home builders site or at the sales office, a lead is upgraded to the prospective home buyer. The prospect is still able to do everything a lead can do. In addition, the prospect can communicate directly and book appointments with a sales representative.

1. Direct communication with a sales representative

2. Book appointments with a sales representative

8.1.4 Under Contract

After signing a sales contract, a customer under a contract has all the same advantages as a prospect has. Furthermore, after the construction of a new house has begun, up-to-date information on the status of the construction process is shown in a customer portal. For example, "Framing has been completed." An under contract is able to see contact information for the people responsible for the building process such as a design representative, superintendent, and quality assurance inspector to name a few.

1. Up-to-date on the status of construction.

2. See contact information for the people responsible for the building process

8.1.5 Homeowner

A homeowner has all the communication methods as a customer under contract. However, since the home is built, the status of construction is needless to show anymore. A customer portal provides a complete history of a home, including the vendors, and selected options for a model house. The homeowner can view home information and warranties.

The customer can submit a service request with an ability to list the actual service request items and post photos for each. They are able to see the status of service request progress and the appointment for the service request inspection time.

1. View home information
2. Submit service request
3. Save tenant information
4. Save customer information

8.2 Design and Implementation

The requirements are done in several phases. The first phase includes requirements related to general user requirements, Lead, and Home Owner. At the moment of writing this thesis, only the first phase has been done. The implemented Web API are in Appendix A.1. The rest of the requirements will be implemented as a future work.

8.2.1 Web API Architecture Style

Web APIs were implemented by Richardson's Rest Maturity Model Level 2 introduced in Section 4.1.3. Adding different HTTP verbs limits how to interact with resources. In this thesis, clients operate on resources with GET and POST verbs. ASP.NET Web API framework uses POST verb for both data creation and update. DELETE verb is not used in this thesis. HTTP Status codes are presented in Appendix A.2

8.2.2 Versioning

This thesis implements the Web API for a customer portal the URI way in version 3 as follows:

Version 1: /api/v3/customer/1

The Web API documentation is presented in Appendix A.1.

8.2.3 Security

8.2.3.1 Token Based Authentication with HMAC-SHA256

This thesis is using HMAC-SHA256 to authenticate the userID and expiration time of the token. With userID 1 and expiration time[1] of 8 hours and an HMAC = TlaOGgBNu((the example token would be 1,8,TlaOGgBNu((where each part is separated with ‘,’-character. The token is provided in each client request as a URL query parameter. The server authenticates the token before operating on a requested resource. The token-based authentication with given example is depicted in Figure 8.1.

[1]Base64 is used for encoding the expiration time for inclusion in URLs.

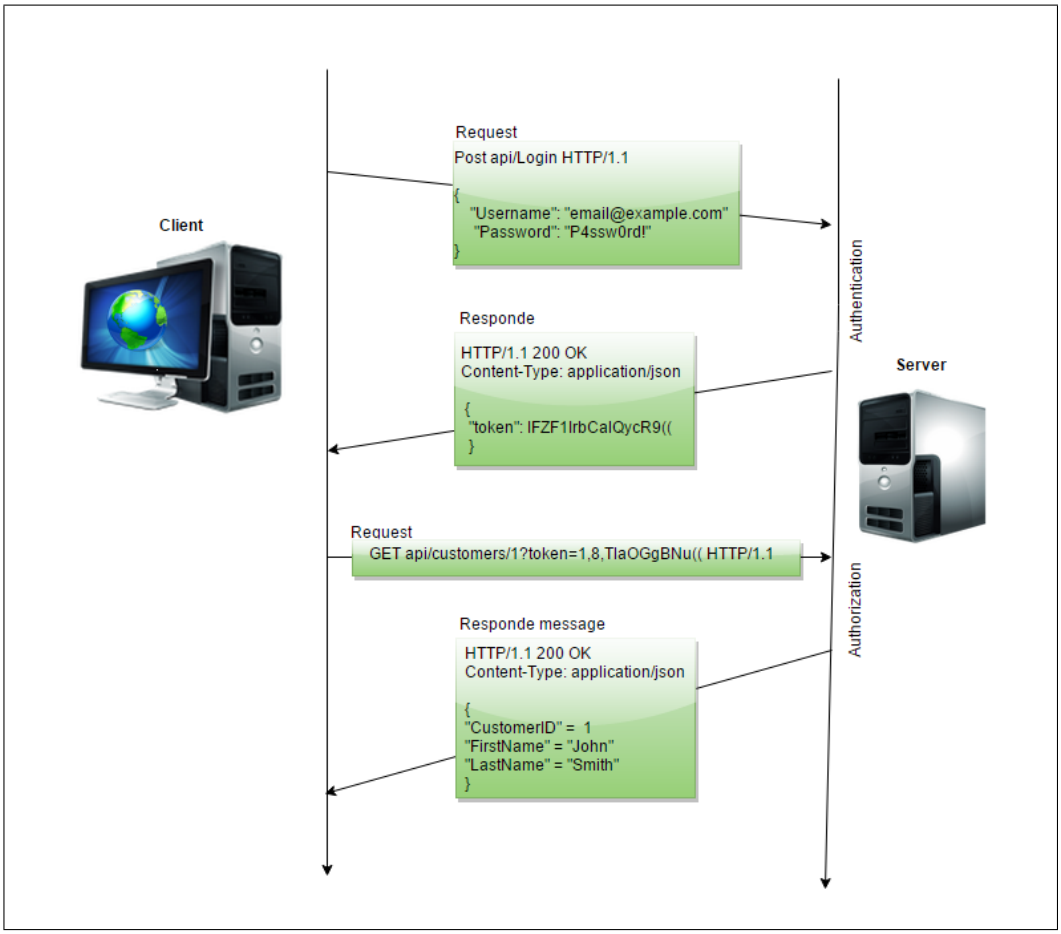

Figure 8.1: Token based authentication and authorization.

8.2.3.2 Posting and Validation

ASP.NET Web API uses attributes to set validation rules for properties [68]. Data validation attributes are implemented for save tenant and save customer info as follows:

```
public class Tenant
{
    [Required]
    public string HomeID { get; set; }
    [Required]
    public string Name { get; set; }
    [Required]
    [ValidPhoneNumber]
    public string Phone { get; set; }
}
public class CustomerInfo
{
    [Required]
    public string HomeID { get; set; }
    public string StreetAddress { get; set; }
    public string City { get; set; }
    [ValidZipCode]
    public string ZipCode { get; set; }
    [ValidStateCode]
    public string StateCode { get; set; }
    [ValidPhoneNumber]
    public string PhoneHome { get; set; }
}
```

The “Required” attribute says that the “HomeID”, “Name” and “Phone”-properties must not be null. The “ValidZipCode” and “ValidPhoneNumer” attributes determine that the properties have to be in certain valid forms. In addition, it is validated that no null or empty “”-characters are saved in the update.

A client can also include additional properties into a JSON. In case the added property does not exist in the model, the value is ignored. Over-posting is a problem if the model consists of read-only properties that are not supposed to be modified. This thesis uses the workaround suggested in Section 6.5.3.

Chapter 9

Result and Discussion

This chapter evaluates the requirements of the web APIs for a customer portal. In addition, the encountered challenges and problems are represented and finally a future work is proposed.

9.1 Evaluation and Challenges

This section evaluates and discusses encountered problems and challenges in implementing Web API for a customer portal.

9.1.1 Architectural Style

Hypermedia as the Engine of Application State (HATEOAS) is a constraint of the REST architectural style. According to Section 4.1.4 a customer portal Web APIs implemented are not RESTful. They do not fulfill Richardson REST Maturity Model Level 3: HATEOAS, which is one of the requirements for the RESTful architectural style.

As Fielding stated: “software design on the scale of decades: every detail is

intended to promote software longevity and independent evolution. Many of the constraints are directly opposed to short-term efficiency. Unfortunately, people are fairly good at short-term design, and usually awful at long-term design" [31].

With HATEOAS, the software can evolve independently over a decade. In our design, it does not need to be RESTful in the HATEOAS way. HATEOAS enhances discoverability and makes an application more open to other application. However, Web APIs for a customer portal are only for our customer Web developers and are not designed to be public. In addition, the HATEOAS constraint is not so practical in real life, since the number of different states may explode in more complicated systems.

9.1.2 Versioning

Chapter 5 presented different version types. The URL way is as follows:

```
Version 1: /api/v1/customer/1
```

and the Hypermedia way in accept headers:

```
GET /api/customer/1 HTTP/1.1
Accept: application/json; version=1
```

The URI way is a simple way for versioning an API. The advantage of the URI versioning is that clients know which version they are using. The drawback of the URI way is that it is not RESTful since different versions of URIs still link to the same resource entity [7]. Another drawback is that the older API versions need to be maintained also and this may cause overhead in Web API development [30].

The hypermedia way of versioning in accept headers versions the representation. However, it is a more complicated approach and it is also more difficult to test. Instead of introducing a new version number in a URL with few characters, a developer has to add a relatively high number of line of codes to address the right version of a representation of a resource [13]. Clients cannot test straightforwardly via a URI. Instead, they have to construct a request and configure an appropriate accept header [15].

From these two ways, the URI versioning is for both the client and the server side developer easier to consume.

Section 5.1 presented different version cases. Adding new fields to a representation did not break a Web API contract. Whenever there is a renaming or removing fields of an existing representation, it is a breaking change and should be introduced in the next version. Changing a resource entity should introduce an entirely new URI.

The drawback in versioning is that it tightly couples to the evolution policies of the Web API providers [30]. Clients are forced to update to new versions to avoid breaking changes in their Web applications. One solution recommended is Web APIs should not be changed too often [30]. The representation should be as minimal as possible since it is easier to add new properties than remove old ones [20]. One important observation is that the early versions of Web APIs are invariably unstable and change-prone [30]. In this thesis, the process is done in cooperation with client developers in small iterations asking feedback until the requirements are met. In addition, an example of a Web API implementation has been given to get additional feedbacks.

9.1.3 Security

9.1.3.1 Token Based Authentication with HMAC-SHA256

Since, no information about the user is stored on the server or in a session, token based authentication is stateless. [65]. Stateless authentication allows the application to scale. Furthermore, token based authentication decouples the client and the server. Authentication and authorization are handled only on the server side. In addition, the token consists of expiration time, which is an important part of the design.

Token based authentication uses HMAC-SHA256. SHA-256 is cryptographically a stronger hash function than MD5 or SHA-1 [26] [21]. It is not recommended to use MD5 and SHA1 since they have known security vulnerabilities [21] and they are already broken in several studies [38] [66]. Although, SHA-1, till date, is still the most widely used hash function, in spite of several successful cryptanalytic attacks against it [38]. However, the attacks remain impractical due to high computation complexity and associated cost [38].

There is no known SHA-256 break yet.

9.1.3.2 Posting and Validation

The implemented Class Tenant and CustomerInfo still have an "Under-posting"-problem. Under-posting happens when some properties are left out. For example, suppose that the client is updating a CustomerInfo where StreetAddress or City-properties are left out. In a model binding, it is assigned a default value of empty character "" for missing properties. This is a problem when updating [68]. In Section 6.5.2 it was suggested to use "?"-character after variable type as "not set". This was not implemented, but can be implemented as a future work. The current implementation only validates that no null or empty ""-properties are updated.

In addition, a safe workaround for POST verb in new entity creation has not been implemented. If during a POST creation task, the request time-outs, there is no way to know has the request reached the server or not. This needs to be taken into consideration in future work.

9.2 Future work

In the future, we are going to implement the rest of the Web API requirements and a safe workaround for POST verb when creating a new instance. Also an "Under-posting"-problem needs to be taken into consideration in future Web API implementations.

Although, it was not possible to implement the pure RESTful Web API with HATEOAS constraint. The idea of loose coupling and independent evolving of the client and the server without breaking changes in Web API is attractive. It is possible to implement HATEOAS in some small Web API scenarios, where the state of actions is small and limited. If the HATEOAS is used then no versioning is needed, just change the URIs in a representation of a resource.

In addition, we are going to maintain and evolve web APIs and build our own customer portal as a prototype for introducing customer portal Web APIs.

Chapter 10

Conclusions

The purpose of this thesis is to design a Web-based Application Programming Interface (Web API) for a customer portal in the home building industry. The implementation of Web APIs allows the home provider to brand their own customer portal independently from the server side. Web developers can use Web APIs as a building element in their Web applications.

This thesis presents a design of a Web API. At the center of the design is the architectural design of Web API including versioning, and security of Web APIs. The goal of this thesis is to provide such a design that both client side and server side can evolve independently.

With implemented Web APIs, homebuilders can customize their Web site. Loose coupling was not gained fully, due to versioning in URIs that can force clients to update to the next version to avoid breaking changes in their applications.

In the future, we are going to implement the rest of the Web API requirements. In addition, we are going to maintain and evolve Web APIs and build our own customer portal as a prototype for introducing pustomer portal Web APIs.

Bibliography

[1] A Guide to REST and API Design. CA Technologies http://transform.ca.com/rest-api-design-guide.html. Accessed: 2015-08-31.

[2] API Design 201: Web API Architectural Styles. API Academy http://www.apiacademy.co/resources/api-design-201-web-api-architectural-styles/. Accessed: 2015-10-20.

[3] ASP.NET. ASP.NET http://www.asp.net/web-api, Security http://www.asp.net/web-api/overview/security/individual-accounts-in-web-api. Accessed: 2015-05-13.

[4] Definiton of Business Logic. https://www.techopedia.com/definition/27382/business-logic. Accessed: 2015-10-20.

[5] Discussions with Antti Tuomi. 2015-07-30.

[6] Discussions with Jouko Väkiparta. 2015-08-21.

[7] eMag Web APIs: From Start to Finish. http://www.infoq.com/articles/Web-APIs-From-Start-to-Finish. Accessed: 2015-11-07.

[8] Introduction to REST and .net Web API. http://blogs.msdn.com/b/martinkearn/archive/2015/01/05/introduction-to-rest-and-net-web-api.aspx.

[9] JSON. http://www.json.org/. Accessed: 2015-05-13.

[10] Kova Finland. http://www.kovafinland.com/. Accessed: 2015-08-07.

[11] Sapphire Build. MiTek `http://www.kovasolutions.com/ProductModules/SapphireBuildOverview.aspx`. Accessed: 2015-08-07.

[12] Understanding HATEOAS. `https://spring.io/understanding/HATEOAS`. Accessed: 2015-10-15.

[13] Versioning ASP.NET Web API Services Using HTTP Headers. `https://seroter.wordpress.com/2012/09/25/versioning-asp-net-web-api-services-using-http-headers/`. Accessed: 2015-11-06.

[14] What is a Portal, Really? `http://compnetworking.about.com/od/internetaccessbestuses/l/aa011900a.htm`. Accessed: 2015-08-27.

[15] Your API versioning is wrong, which is why I decided to do it 3 different wrong ways. `http://www.troyhunt.com/2014/02/your-api-versioning-is-wrong-which-is.html`. Accessed: 2015-11-15.

[16] Iso/iec/ieee systems and software engineering – architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)* (Dec 2011), 1–46.

[17] MiTek Acquires Kova Solutions, LLC. Business Wire `http://www.businesswire.com/news/home/20131010005121/en/MiTek-Acquires-Kova-Solutions-LLC#.VcRWMvntlBc`, October 2013. Accessed: 2015-08-07.

[18] Aihkisalo, T., and Paaso, T. Latencies of service invocation and processing of the rest and soap web service interfaces. In *Services (SERVICES), 2012 IEEE Eighth World Congress on* (June 2012), pp. 100–107.

[19] Andrews, G. R. Paradigms for process interaction in distributed programs. *ACM Comput. Surv. 23*, 1 (Mar. 1991), 49–90.

[20] Block, G., Cibraro, P., Felix, P., Dierking, H., and Miller, D. *Designing Evolvable Web APIs with ASP.NET*, 1st ed. O'Reilly Media, Inc., 2014.

[21] Boyles, T. *CCNA Security Study Guide: Exam 640-553*. SYBEX Inc., Alameda, CA, USA, 2010.

[22] Bureau, U. S. C. New Privately Owned Housing Units Started. `https://www.census.gov/construction/nrc/pdf/startsan.pdf`. Accessed: 2015-08-21.

[23] C., C. The self-service industry's best and worst of 2009. `http://www.kioskmarketplace.com/articles/the-self-service-industrys-best-and-worst-of-2009/`. Accessed: 2015-10-05.

[24] Caulfield, J. Software Program Allows Builders to Operate from Single Database. `http://www.builderonline.com/building/operations/software-program-allows-builders-to-operate-from-single-database_o`, September 2011. Editor: Builder, Accessed: 2015-07-24.

[25] Chang, D., Gupta, K. C., and Nandi, M. Rc4-hash: A new hash function based on rc4. In *Proceedings of the 7th International Conference on Cryptology in India* (Berlin, Heidelberg, 2006), INDOCRYPT'06, Springer-Verlag, pp. 80–94.

[26] Cheng, N., Wang, Y., Zhao, X., and Li, N. The digital fingerprint of xml electronic medical records based on hmac-sha256 algorithm. In *Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on* (May 2011), pp. 338–340.

[27] Costello, R. L. Building Web Services the REST Way. `http://www.xfront.com/REST-Web-Services.html`.

[28] Dang, Q. Changes in federal information processing standard fips 180-4, secure hash standard. *Cryptologia 37*, 1 (Jan. 2013), 69–73.

[29] Ebert, J. SOA with REST: Principles, Patterns & Constraints for Building Enterprise Solutions with REST by Thomas Erl, Benjamin Carlyle, Cesare Pautasso, Raj Balasubramanian. *SIGSOFT Softw. Eng. Notes 38*, 3 (May 2013), 32–33.

[30] Espinha, T., Zaidman, A., and Gross, H.-G. Web api growing pains: Stories from client developers and their code. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on* (Feb 2014), pp. 84–93.

[31] Fielding, R. T. Roy Fielding on REST APIs must be hypertext-driven. `http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven`. Accessed: 2015-05-13.

[32] Fielding, R. T. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, 2000. AAI9980887.

[33] Fielding, R. T., and Taylor, R. N. Principled design of the modern web architecture. *ACM Trans. Internet Technol. 2*, 2 (May 2002), 115–150.

[34] Flanders, J. *RESTful .NET: Build and Consume RESTful Web Services with .NET 3.5*. O'Reilly Media, Inc., 2008.

[35] freshome. Design & Architecture Magazine, 10 Basic Facts You Should Know About Modular Homes. `http://freshome.com/2013/03/27/10-basic-facts-about-modular-homes/`. Accessed: 2015-07-28.

[36] Fu, C., Belqasmi, F., and Glitho, R. Restful web services for bridging presence service across technologies and domains: an early feasibility prototype. *Communications Magazine, IEEE 48*, 12 (December 2010), 92–100.

[37] Giridhar, C. Maturity Model of Web Services. QCon talk `https://technobeans.wordpress.com/2012/09/12/maturity-model-of-web-services/`, September.

[38] Hassan, M., Khalid, A., Chattopadhyay, A., Rechberger, C., Guneysu, T., and Paar, C. New asic/fpga cost estimates for sha-1 collisions. In *Digital System Design (DSD), 2015 Euromicro Conference on* (Aug 2015), pp. 669–676.

[39] HiP. What is a Lead? What is a Prospect? What's the Difference? `http://high-impact-prospecting.com/what-is-a-lead-what-is-a-prospect-whats-the-difference/`. Accessed: 2015-07-29.

[40] Hyväri, P. 3D Visualization of Configured Homes. Master's thesis, Department of Computer and Science and Engineering, Aalto University School of Science, Espoo, Finland, 1 2007.

[41] IBISWorld. Home Builders in the US: Market Research Report. `http://www.ibisworld.com/industry/default.aspx?indid=169`. Accessed: 2015-07-24.

[42] Jacobson, D., Woods, D., and Brail, G. *APIs: A Strategy Guide*. Oreilly and Associate Series. O'Reilly Media, 2011.

[43] Jon Flanders. More On REST. `https://msdn.microsoft.com`. Accessed: 2015-09-02.

[44] Joudeh, T. Building ASP.Net Web API RESTful Service. `http://bitoftech.net/2013/12/16/asp-net-web-api-versioning-accept-header-query-string`. Accessed: 2015-05-13.

[45] Krawczyk, H., Bellare, M., and Canetti, R. HMAC: Keyed-Hashing for Message Authentication. RFC 2104 (Informational), February 1997.

[46] Kurtz, J., and Wortman, B. *ASP.NET Web API 2: Building a REST Service from Start to Finish*, 2nd ed. Apress, Berkely, CA, USA, 2014.

[47] Laplante, P. A. *What Every Engineer Should Know About Software Engineering (What Every Engineer Should Know)*. CRC Press, Inc., Boca Raton, FL, USA, 2007.

[48] Line, V. Industry Analysis: Homebuilding. `http://www.valueline.com/Stocks/Industries/Industry_Analysis__Homebuilding.aspx#.VdXH_PntlBc`. Accessed: 2015-08-15.

[49] Mulligan, G., and Gracanin, D. A comparison of soap and rest implementations of a service based interaction independence middleware framework. In *Simulation Conference (WSC), Proceedings of the 2009 Winter* (Dec 2009), pp. 1423–1432.

[50] NAHB. Housing Market Index (through August 2015). `http://www.nahb.org/`. Accessed: 2015-08-21.

[51] NAHB. Production Homes. `http://www.nahb.org/en/consumers/home-buying/types-of-home-construction/types-of-construction-production-homes.aspx`. Accessed: 2015-07-24.

[52] Nath, S. Web services: Design choices for space ground system integration. In *MILITARY COMMUNICATIONS CONFERENCE, 2012 - MILCOM 2012* (Oct 2012), pp. 1–6.

[53] National Institute of Standards and Technology. *FIPS PUB 180-2: Secure Hash Standard*. 2004.

[54] NewHomeSource. Is a Custom or Production Home Builder Right for You? `http://www.newhomesource.com/resourcecenter/articles/is-a-custom-or-production-builder-right-for-you`. Accessed: 2015-07-28.

[55] Pagni, M., Hau, J., and Stockinger, H. A multi-protocol bioinformatics web service: Use soap, take a rest or go with html. In *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on* (May 2008), pp. 728–734.

[56] Pautasso, C., Zimmermann, O., and Leymann, F. Restful web services vs. "big"' web services: Making the right architectural decision. In *Proceedings of the 17th International Conference on World Wide Web* (New York, NY, USA, 2008), WWW '08, ACM, pp. 805–814.

[57] Philip, N. Empowering the customer of the future through self serving web portals-using www.truvolounge.ie as a case study. `http://trap.ncirl.ie/507/`, 2010.

[58] Potlapally, N., Ravi, S., Raghunathan, A., and Jha, N. A study of the energy consumption characteristics of cryptographic algorithms and security protocols. *Mobile Computing, IEEE Transactions on 5*, 2 (Feb 2006), 128–143.

[59] Programmable Web. `http://www.programmableweb.com/`, Most Popular APIs`http://www.programmableweb.com/news/most-popular-apis-least-one-will-surprise-you/2014/01/23`. Accessed: 2015-05-13.

[60] Ramanathan, R. *Handbook of Research on Architectural Trends in Service-Driven Computing*. Advances in Systems Analysis, Software Engineering, and High Performance Computing:. IGI Global, 2014.

[61] Richardson, L. Justice Will Take Us Millions Of Intricate Moves. QCon talk `http://www.crummy.com/writing/speaking/2008-QCon/`, 2008.

[62] Rivest, R. The md5 message-digest algorithm, 1992.

[63] Song, Y., Xu, K., and Liu, K. Research on web instant messaging using rest web service. In *Web Society (SWS), 2010 IEEE 2nd Symposium on* (Aug 2010), pp. 497–500.

[64] Strother, J., Fazal, Z., and Rettich, K. From full-service to self-service: The airline industry takes off. In *Professional Communication Conference (IPCC), 2010 IEEE International* (July 2010), pp. 191–194.

[65] W3C. Web Service `http://www.w3.org/TR/ws-gloss/`, SOAP `http://www.w3.org/TR/soap12/`, HTTP/1.1 `http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10`, Hypertext `http://www.w3.org/WhatIs.html`, Token Based Authentication `http://www.w3.org/2001/sw/Europe/events/foaf-galway/papers/fp/token_based_authentication/`. Accessed: 2015-11-05.

[66] Wang, X., Yin, Y. L., and Yu, H. Finding collisions in the full sha-1. In *Advances in Cryptology - CRYPTO 2005: 25th Annual International*

Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings (2005), vol. 3621 of *Lecture Notes in Computer Science*, Springer, pp. 17–36.

[67] Washington, M., and Lackey, I. *Building Websites with DotNetNuke 5*. Packt Publishing, 2010.

[68] Wasson, M. Model Validation in ASP.NET Web API. `http://www.asp.net/web-api/overview/formats-and-model-binding/model-validation-in-aspnet-web-api`. Accessed: 2015-05-13.

Appendix A

Web API Documentation and HTTP Status Codes

General HTTP response status codes for Web APIs are already introduced in Section A.2 they will not be shown in the Web API documentation. The posts and the responses are returned in JSON.

A.1 Web API Documentation

A.1.1 General

Login	
URI	api/v3/Customer/Login
HTTP verb	POST
Body	[Required] string Email [Required] string Password
Success	string token=userID;expiration_time;HMAC-SHA256

Email Password Reset Link

URI	api/v3/Customer/EmailPasswordResetLink?token={token}
HTTP verb	POST
Body	[Required] string Email
Success	string resetURL = http://example.com/ResetPassword.html? PasswordResetToken={GeneratedPasswordResetToken}

Reset Password

URI	api/v3/Customer/ResetPassword?token={token}
HTTP verb	POST
Body	[Required] string NewPassword [Required] string NewPasswordConfirm
Success	-

Change Password

URI	api/v3/Customer/ChangePassword?token={token}
HTTP verb	POST
Body	[Required] string OldPassword [Required] string NewPassword [Required] string NewPasswordConfirm
Success	-

A.1.2 Lead

Print Brochures

URI	/api/v3/Lead/ListBrochures?token={token}
HTTP verb	GET
Body	-
Success	List<LeadBrochure> string Name string Description

Print Brochure

URI	/api/v3/Lead/PrintBrochure/[LeadBrochureID]?token={token}
HTTP verb	GET
Body	[Required] int LeadBrochureID
Success	int LeadBrochureID string URLforBrochure

A.1.3 Home Owner

Get Homes

URI	/api/v3/Home/ListHome?token={token}
HTTP verb	GET
Body	-
Success	List<string>HomeID

Submit Service Request

URI	/api/v3/Home/SaveServiceRequest?token={token}
HTTP verb	POST
Body	[Required] string HomeID [Required] string Description [Required] string Name List<ServiceRequestItem>
Success	List<ServiceRequest>

View Home Information

URI	api/v3/Home/GetInfo/homeID?token={token}
HTTP verb	GET
Body	[Required] string HomeID
Success	string Name string WarrantyStatus date WarrantyExpirationDate BuyerInfo Community Lot Model List<SelectedHomeOptions>

Save Tenant Information

URI	/api/v3/Home/SaveTenant?token={token}
HTTP verb	POST
Body	[Required] string HomeID [Required] string Phone [Required] [ValidPhoneNumber] string Name
Success	-

Save Customer information

URI	/api/v3/Home/SaveCustomerInfo?token={token}
HTTP verb	POST
Body	[Required] string HomeID string StreetAddress string City [ValidZipCode] string ZipCode [ValidStateCode] string StateCode [ValidPhoneNumber] string PhoneHome
Success	-

A.2 HTTP Status Codes

Status Code	**API meaning**
200 OK	Response to a successful GET, PUT or DELETE
201 Created	Response to a POST that a resource is created successfully
204 No Content	Response to a DELETE, where a successful request won't be returning a body
400 Bad Request	The request parameters are not valid
401 Unauthorized	When an authentication is failed
403 Forbidden	When an authentication succeeded but authenticated user's access to the resource is forbidden
404 Not Found	When a requested resource does not exits

410 Gone	Indicates that the resource at this end point is no longer available. Useful as a response for old API versions
500 Internal Server Error	The server encountered an unexpected condition
501 Not Implemented	The service is not implemented
503 Service Unavailable	The service is unavailable due to maintenance